Generating a Diversity of Challenging Programming Puzzles with Autotelic Generative Models

Julien Pourcel,
Inria

Cédric Colas
MIT, Inria

Gaia Molinaro
University of California, Berkeley

Pierre-Yves Oudeyer
Inria

Laetitia Teodorescu
Inria

Abstract

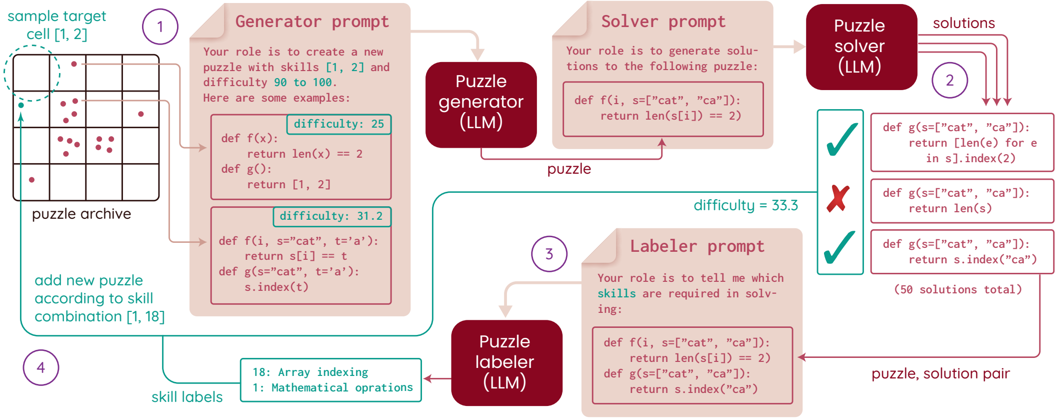
The ability to invent novel and interesting problems is a remarkable feature of human intelligence that drives innovation, art, and science. We propose a method that aims to automate this process by harnessing the power of state-of-the-art generative models to produce a diversity of challenging yet solvable problems, here in the context of Python programming puzzles. Inspired by the intrinsically motivated literature, Autotelic CodE Search (ACES) jointly optimizes for the diversity and difficulty of generated problems. We represent problems in a space of LLM-generated semantic descriptors describing the programming skills required to solve them (e.g. string manipulation, dynamic programming, etc.) and measure their difficulty empirically as a linearly decreasing function of the success rate of *Llama-3-70B*, a state-of-the-art LLM problem solver. ACES iteratively prompts a large language model to generate difficult problems achieving a diversity of target semantic descriptors (goal-directed exploration) using previously generated problems as in-context examples. ACES generates problems that are more diverse and more challenging than problems produced by baseline methods and three times more challenging than problems found in existing Python programming benchmarks on average across 11 state-of-the-art code LLMs.

1 Introduction

Humans are not only talented problem solvers, they are first of all remarkable *problem generators* — generating endless streams of new problems for themselves and others Chu and Schulz [2020], Molinaro and Collins [2023]. We set build problems for others to learn, set challenges to ourselves, [Burton and Hiron, 2008], aggregate problems to train and test AI models [Hendrycks et al., 2020, Chen et al., 2021], and come up with new problems that drive innovation in art and science [Gromov, 2018, Chu et al., 2024]. This intrinsic drive to generate problems for oneself — the *autotelic property* — has further been argued to drive the capacity for adaptation and open-ended learning in both human [Chu and Schulz, 2020] and machines [Schmidhuber, 2013, Herrmann et al., 2022, Colas et al., 2022]. Automating this problem-generation process would have numerous positive applications: for instance, designing exercises tailored to optimize the learning experience of every human or machine learner (automatic curriculum learning Portelas et al. [2020]); or facilitating the generation of evaluation protocols (human tests, machine learning benchmarks). It would provide the necessary curriculum for open-ended learning machines [Colas et al., 2022] and may be a key component of automated scientific discovery [Grizou et al., 2020, Etcheverry, 2023].

We propose to leverage machine learning — a set of tools usually targeted at *solving* problems — to automate the *generation of a diverse set of interesting problems*, here in the domain of Python programming puzzles. Programming puzzles indeed represent an open-ended space of problems to

38th Conference on Neural Information Processing Systems (NeurIPS 2024).



Your role is to create a new puzzle with skills [1, 2] and difficulty 90 to 100. Here are some examples:

difficulty: 25

```
def f(x):
    return len(x) == 2
def g():
    return [1, 2]
```

difficulty: 31.2

```
def f(i, s="cat", t='a'):
    return s[i] == t
def g(s="cat", t='a'):
    s.index(t)
```

Your role is to generate solutions to the following puzzle:

```
def f(i, s=["cat", "ca"]):
    return len(s[i]) == 2)
```

```
def g(s=["cat", "ca"]):
    return [len(e) for e
     in s].index(2)
```

```
def g(s=["cat", "ca"]):
    return len(s)
```

```
def g(s=["cat", "ca"]):
    return s.index("ca")
```

(50 solutions total)

Your role is to tell me which skills are required in solving:

```
def f(i, s=["cat", "ca"]):
    return len(s[i]) == 2)
def g(s=["cat", "ca"]):
    return s.index("ca")
```

```
18: Array indexing
1: Mathematical oprations
```

Figure 1: **Overview of the ACES algorithm.** ACES iteratively generates a diverse set of challenging programming puzzles. First, a target *cell* corresponding to a combination of programming skills is sampled from a puzzle archive (1), and puzzles from filled neighboring cells — prioritized by difficulty — are selected as examples and given to a puzzle generating LLM. It generates a new puzzle with the desired skill combination that an LLM solver tries to solve 50 times (2). If never solved, the puzzle is discarded. An LLM describes the skills needed to solve the puzzle (3) and the puzzle, along with its computed difficulty score, is added to the puzzle archive.

explore: from simple string manipulations to complex dynamic programming or open mathematical puzzles [Schuster et al., 2021]. We qualify problems as *interesting* when they are challenging yet solvable. This can be estimated by computing the empirical difficulty of a puzzle for a particular solver: out of 50 attempts, the solver should solve the problem at least once (solvability) but as rarely as possible (difficulty). In contrast with natural language instruction domains, our puzzle domain lets us objectively check the validity of a given solution by simply running a Python interpreter. This domain thus affords both an open-ended space to explore (diversity search) and an objective quality measure to maximize.

The standard approach for problem generation simply queries pretrained generative models with few-shot examples or specific instructions [Haluptzok et al., 2022, Honovich et al., 2023, Gunasekar et al., 2023]. This amounts to sampling from a stationary distribution such that the quality and novelty of generated problems reflect those of the problems found in the training data. Instead, we introduce *Autotelic CodE Search* (ACES), an *autotelic generative model* that steers pretrained generative models to produce a diversity of challenging puzzles by iteratively reusing previously generated puzzles as examples to guide the production of *newer and harder problems*.

ACES builds on *Map-Elites* [Mouret and Clune, 2015], an evolutionary quality-diversity (QD) method [Pugh et al., 2016]. In all QD algorithms, the user first needs to define a *descriptor function* mapping each generated outcome (here, each problem) to a numerical representation that will be used to measure diversity. Programming puzzles are high-dimensional objects that we chose to represent by the *set of programming skills required to solve them* — a high-level semantic description space that better captures intuitive notions of puzzle diversity than pretrained embedding representations. We obtain these by asking a large language model (LLM) to label each problem a set of skills from a list of 20 possible ones. Just like Map-Elites, ACES maintains an archive of generated outcomes grouped by skill sets (descriptor niches) and optimizes quality (here difficulty) locally within each niche. While Map-Elites randomly mutates solutions sampled from the archive in hopes of discovering a new niche or finding higher-performing solutions, ACES performs an *explicit goal-directed exploration* [Colas et al., 2022]. At each iteration, ACES targets a goal descriptor, carefully selects relevant and challenging example problems from the archive and, conditioned on these, prompts an LLM to generate a more difficult problem-solution pair achieving the goal. The new puzzle is labelled, evaluated and, if valid, is added to its corresponding descriptor niche in the archive. Across iterations, the archive gets filled with more diverse and challenging puzzles which provide higher quality examples to guide the generation of yet more diverse and challenging puzzles (see Figure 1).

We show that ACES generates a wider diversity of more challenging problems than both standard generative approaches [Haluptzok et al., 2022, Honovich et al., 2023] and existing algorithms based on Map-Elites [Bradley et al., 2023a]. Finally, we show that generated problems are harder to solve than those found in existing Python programming benchmarks, this for all the 11 state-of-the-art LLM-based problem solvers we tested. Whereas the HumanEval+ benchmark is starting to saturate (*GPT-4-turbo* achieves a pass@1 of 86.6%, *CodeQwen1.5-7B-Chat* 78.7%, [Liu et al., 2024])[1], the best solver (*Mixtral-8x22B-Instruct-v0.1*) only achieves a pass@1 of 47.3% on our problems. This work paves the way for automating the design of harder benchmarks whose difficulty is calibrated using LLM solvers themselves, eventually allowing evaluations to increase in difficulty as models improve.

2 Related Work

Open-ended exploration algorithms Our puzzle-generating method is at the intersection of two research lines: evolutionary computing [Lehman and Stanley, 2011a,b, Mouret and Clune, 2015, Pugh et al., 2016, Cully and Demiris, 2018a, Lehman et al., 2022] and intrinsically-motivated learning [Baranes and Oudeyer, 2013, Etcheverry et al., 2020, Forestier et al., 2022, Colas et al., 2022]. Beginning with *novelty search* [Lehman and Stanley, 2011a,c], the evolutionary approach to exploration expanded with the invention of quality-diversity algorithms [QD: Lehman and Stanley, 2011b, Mouret and Clune, 2015, Cully and Demiris, 2018a], a set of methods striving to evolve a diverse population of locally-performant solutions via the undirected mutation of existing solutions. A parallel line of research introduced *goal-directed* exploration processes, also called *autotelic learning*, where agents learn to represent and sample their own goal as a way to direct the diversity search [Baranes and Oudeyer, 2013, Forestier et al., 2022, Colas et al., 2022]. Although autotelic methods were first developed to model the open-ended development of children in skill learning robots Baranes and Oudeyer [2013], Moulin-Frier et al. [2014], Oudeyer and Smith [2016], they also proved effective in the automatic exploration of complex systems, either simulated [Reinke et al., 2019, Etcheverry et al., 2020] or physical [Grizou et al., 2020].

LLM-augmented exploration LLMs can be useful in various parts of quality-diversity and exploration algorithms. Their capacity to generate appropriate variations of existing text has been used to implement mutation [ELM: Lehman et al., 2022] and crossover [Meyerson et al., 2023] operators within QD, with applications to neural architecture search [Chen et al., 2023, Nasir et al., 2023]. Following the recent trend in learning from AI feedback [Bai et al., 2022, Lee et al., 2023] and using LLM-as-judge methods [Zheng et al., 2024], recent work has used LLM responses as models of *interestingness* [Zhang et al., 2023, Klissarov et al., 2023, Sachdeva et al., 2024] within RL, while others have augmented Map-Elites with LLM-based quality judgments and semantic descriptors for creative writing [Bradley et al., 2023b] or adversarial prompt generation [Samvelyan et al., 2024]. These last two are close to our ELM baseline, but in contrast to these works we use a grounded empirical difficulty metric as quality compared to AI feedback which is less grounded and might be innacurate. ACES additionally uses LLM-augmented goal generation, which echoes recent methods leveraging generative models for open-ended goal-based exploration [Colas et al., 2023, Wang et al., 2023a, Du et al., 2023]. In contrast to these works we optimize for both the difficulty and the diversity of the generated puzzles.

LLMs for generating code and instruction datasets This work is also linked to prior approaches for generating synthetic code and instruction data, mostly in data augmentation contexts. Seminal works leveraged a standard generative approach by prompting an LLM to generate new problems using as few-shot examples problems sampled from an existing dataset [Haluptzok et al., 2022, Honovich et al., 2023, Roziere et al., 2023], or problems generated at previous iterations [Wang et al., 2023b]. Closer to the goal-targeting of ACES, Eldan and Li [2023] generates diverse training data by asking an LLM to write stories employing a combinations of words randomly sampled from a large list. Gunasekar et al. [2023], Abdin et al. [2024] build upon this approach to generate programming textbooks by mixing subjects and audiences, and generates exercises by randomizing the exercise name. *Evol-Instruct* is an evolutionary method that iteratively generates language or code instructions by applying prompts that modify previously generated problems by, among other things, increasing their difficulty [Xu et al., 2023, Luo et al., 2023]. They do not optimize for diversity and do not use actual difficulty measurements, relying on the LLM's problem modification to increase it. Finally, *Skill-Mix* [Yu et al., 2023] generates language evaluations for LLMs by generating problems involving combinations of skills in the language domain and grading models using GPT-4; they do not use

---

[1] https://evalplus.github.io/leaderboard.html

```
def f(ls: List[str]):
    """Divide the decimal representation of 8^88 up into strings of
        length eight."""
    return "".join(ls)==str(8**88) and all(len(s)==8 for s in ls)
def g():
    return [str(8**88)[i:i+8] for i in range(0,80,8)]
assert f(g()) == True
```

Figure 2: **Puzzle example.** A simple programming puzzle and its solution from the P3 dataset [Schuster et al., 2021]. A solution function `g` must return a valid solution such that `f(g()) == True`.

any few-shot examples, do not optimize nor measure difficulty for an LLM while generating their problems.

3 Methods

3.1 Programming puzzles and the P3 dataset

The *Python Programming Puzzles dataset* (P3) contains 1715 puzzle-solution pairs where each puzzle is defined by a short test program `f` designed to verify the validity of solution programs `g` such that valid solutions satisfy `f(g()) == True` when run in an interpreter, see example in Figure 2 [Schuster et al., 2021]. P3 puzzles span problems of various difficulties that involve different programming skills: from classic problems (tower of Hanoi) and string manipulations, to factoring problems, dynamic programming, or even open problems in computer science or mathematics. The P3 dataset is split into training and testing datasets ($N = 636$ and 1079 respectively). Both datasets are pre-filtered to examples shorter than 1024 tokens to accommodate for limited context windows in LLMs.

3.2 Generating diverse sets of challenging puzzles

In this work, we aim to generate sets of puzzles that are collectively as diverse and on average as difficult as possible. In this section, we first define the difficulty metric we use, and then how we quantify diversity.

Empirical puzzle difficulty We measure the empirical difficulty of a puzzle with respect to a target LLM solver as the opposite of the solver's competence on that puzzle. We measure competence using the standard pass@k metric with k=1: the number of valid solutions generated after k=1 attempts (which is simply the success rate) [Chen et al., 2021]. We estimate the pass@1 competence over $N = 50$ solving attempts and report the empirical difficulty (puzzle fitness $\mathcal{F}$) as its negative success rate rescaled to the 0–100 range:

$$\mathcal{F}(\texttt{f}) = \begin{cases} (-\text{pass@1}(\texttt{f}, \text{LLM}) + 1) \times 100 \text{ if pass@1} \neq 0 \\ -\infty \text{ otherwise} \end{cases} \tag{1}$$

The more difficult the puzzle is, the higher its fitness. Puzzles for which no solution is found are considered invalid and are discarded. The prompt used for generating solutions can be found in the Appendix. Compared to LLM-based methods of assessing the quality of a sample (typically, critic or LLM-as-judge methods), this difficulty-based metric measures a ground-truth objective: how hard a given puzzle is for a target LLM solver. Our experiments found that this difficulty measure mostly transfers across models: a puzzle that is harder for one model is often harder for others (see Section 4.5). Although our difficulty metric is rather expensive to compute, it captures exactly the intended objective and is thus harder to hack or overfit compared to objectives based on LLM feedback [Zheng et al., 2024, Sachdeva et al., 2024]. This said, an existing possibility for hacking the difficulty metric is to import a random number generator library and to make the `f` function return True with probability 1/50. We have not observed this phenomenon in our experiments.

Skill combination diversity The diversity measure we choose to optimize will have an important impact on the distribution of generated puzzles. Ideally, the set of puzzles should be diverse in their structures and topics. We thus define a set of 20 tags corresponding to different programming skills and label each programming puzzle with the combination of skills needed to solve it (reminiscent of Skill-Mix [Yu et al., 2023]). We then measure *semantic diversity* as the number of unique skill combinations for which there is at least one representative in our generated set. We refer to a tag combination as a *niche* or a *cell* in the rest of the paper. We sample our 20 tags from a larger list generated by an LLM (*GPT-4-0125*) and validated against lists of programming topics covered in classic computer science textbooks and competitive programming platforms (LeetCode and HackerRank). The complete list of tags is given in Appendix Section! A.2 and contains items such as *Recursion*, *Geometry and coordinate problems* or *Hashing*. We limit combinations to a maximum of 5 skills out of 20 possible skills, which gives us 21,700 total niche combinations. This helps avoid unrealistic skill combinations. We prompt an LLM to label puzzles with a set of programming skills as in Bradley et al. [2023b], Samvelyan et al. [2024].

Embedding diversity The diversity measure is based on an LLM's feedback and is thus subject to inaccuracies, especially if this metric is used as an optimization target. As complemetary measures of diversity, we propose *embedding diversity* metrics that estimate diversity in a variety of pretrained embedding spaces that were not used by the algorithms. This measure is computed as the average pairwise cosine distance between the embeddings of all problems generated in the set.

3.3 ACES: Autotelic CodE Search

In this section we present ACES, an exploration method that samples target niches (sets of descriptors) as goals and prompts an LLM with relevant challenging puzzles sampled from an archive of previously generated puzzles to reach them. Like ELM, which we use as baseline, ACES uses an LLM to generate mutations of existing samples from the archive. Compared with ELM, where the LLM is instructed to produce a variation of a given puzzle without a specific objective in mind, ACES instructs the LLM to generate a new puzzle based on its examples. The generated puzzles are then evaluated for fitness and labeled, and added to the archive before the next generation round. The cells are initialized with the deduplicated[2] P3 train set puzzles. An overview of the algorithm is given in Figure 1.

Sampling a goal and relevant examples First, we sample a cell uniformly, then we look at the 3 closest cells for which there is at least one puzzle (the target cell can be one of them). For each of these neighbor cells, we select one puzzle and add it as an example to the prompt. The prompt (available in the Appendix) instructs the LLM to produce puzzles in the target cell (corresponding to a combination of skills) using the 3 sampled puzzles as examples. To select examples from one cell, we normalize the range of fitness scores in this cell between 0 and 1 and then sample from a softmax distribution from these normalized fitnesses with a temperature of 0.2. To guide the model towards generating harder puzzles, we added the difficulty score $\mathcal{D}$ of each puzzle in the prompt (ranging from 0 to 100), instructing the model to reach a score between 90 and 100 when generating new puzzles. The intuitions underlying the design of ACES are: to drive discovery of puzzles with novel skill combinations, we rely on the LLM recombining elements from puzzles close to the target cell along with very often selecting target cells without any representatives. To drive discovery of harder puzzles, we rely on sampling harder puzzles more often as examples and the assumption that harder examples, along with the explicit instruction to create hard puzzles, will lead to more difficult generated puzzles.

Generator and labeler LLMs Once the prompt is built, the generator LLM is instructed to produce five new puzzles following the desired instruction. For each of these generated puzzles, their fitness is computed as the negative success rate over 50 attempts (Equation 1). If the puzzle is not solved by the solver model in 50 attempts, it is considered unsolvable and discarded. For each solvable puzzle, a solution is sampled randomly from the valid ones, and the (puzzle, solution) pair is then described by an LLM. This description is a short text explaining what the puzzle is (like docstrings in P3, or instructions in HumanEval). The (puzzle, solution, description) triplet is then handed to a labeler LLM that produces the skill tags for the puzzle. The description and labeling prompts can be found in Appendix.

---

[2] Every P3 puzzle comes in several instances with different arguments. We randomly sample one instance per puzzle, which gives use 155 seed puzzle instances.

Avoiding label hacking The reason we generate the puzzle description separately from the puzzle itself is that in preliminary experiments, we observed a form of *label hacking* where the generator LLM, when tasked with generating puzzles with a particular combination of skills, generated generic simple puzzles and listed the relevant skills in the description even if they were not required in solving the puzzle. These puzzles were then wrongly tagged with a rare combination of skills and were thus oversampled as examples for the next generations, leading to the propagation of misleading descriptions. Describing the puzzle independently of all references to desired programming skills mitigates this sort of label hacking. It needs to be noted that all forms of AI feedback, be it for the fitness or for the tags, are susceptible to hacking; all methods relying on the optimization of AI feedback signals should consider and mitigate this form of hacking.

3.4 Baselines and variations

ELM with semantic categories We test an ablation of goal-directedness to study its impact, and this is exactly applying ELM to our task by using the same quality and descriptor function. To create new individuals, a cell with at least one representative is sampled at random, an individual is sampled in this cell using the same quality-based sampling mechanism as ACES, and an LLM is instructed to compose a variant of this puzzle using two other random puzzles from the overall archive as few-shot examples of what the puzzle domain looks like. The few-shot examples are given so ELM and ACES both use the same number of examples in their prompts. We term this baseline ELM (in reference to Evolution through Large Models which inspired this work), even if the original ELM did not use any form of AI feedback.We study an additional method combining ACES and ELM. This method also mutates a single puzzle, but the LLM is instructed to produce a targeted variation by trying to reach a target cell like in ACES. We term this method ACES-ELM.

ELM with CVT + embeddings We additionally study the impact of using natural language-based descriptors to measure and optimize for diversity, compared with embeddings. Previous QD methods [Vassiliades et al., 2017] aiming to extend their descriptor spaces to larger dimensions used centroidal Voronoi tessellations [CVT: Du et al., 1999] to partition the space into a tractable number of cells. In the ELM-CVT baseline, we use the P3 train puzzle embeddings as seeds to generate 40000 points in embedding space by adding Gaussian noise with mean 0 and standard deviation 0.12. We then cluster all these points into 10000 clusters which are used as centroids for our Voronoi cells. ELM-CVT behaves as ELM, but the cells are defined with their Voronoi cells instead of programming skill combinations. Embeddings are computed with the *code5p-embedding* model.

Static Gen Our final baseline is a standard generative method using a static few-shot prompting mechanism for puzzle generation, similar to Unnatural Instructions [Honovich et al., 2023] and Haluptzok et al. [2022]. At each time step, 3 puzzles are randomly sampled from the P3 train set and given as examples to an LLM for generating new puzzles. Generated puzzles are not re-used as examples. Puzzle generation prompts for all methods are available in the Appendix.

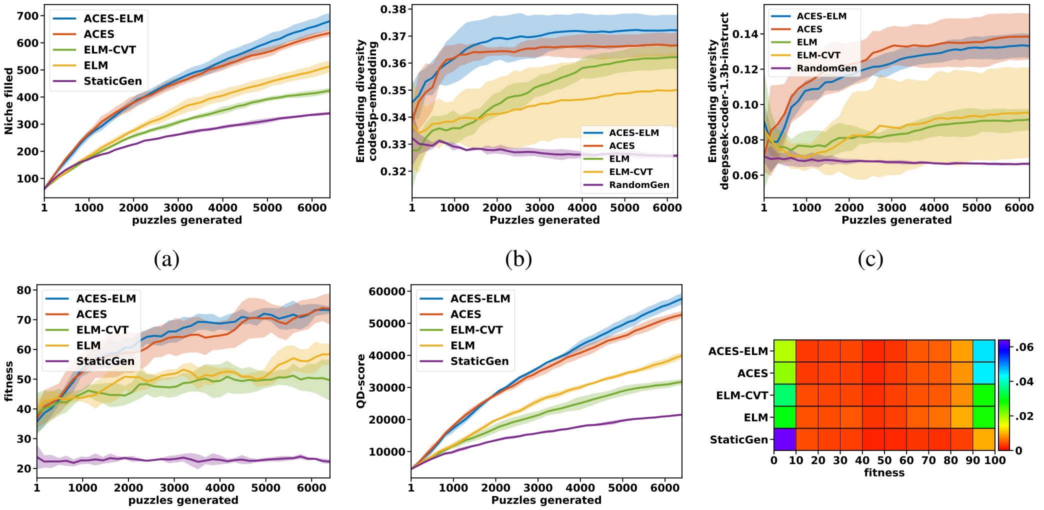
4 Results

4.1 Experimental details

Puzzle generation, solution generation, description generation, and puzzle labeling are all implemented with the state-of-the-art open source model *Llama 3 70B*, quantized in 4 bits, with a temperature parameter of 0.8.[3]. We repeat all experiments using 3 different random seeds and report the means and standard deviations of the results. Each experiment was performed on 1 node of 4 Nvidia Tesla V100 SXM2 32 GB, with 160 GB RAM, for about 20 hours using the vLLM library [Kwon et al., 2023]. Each experiment is run for 40 generations, where each generation corresponds to 160 puzzles generated by the puzzle generator — a total of 6400 puzzle generation attempts per run.

---

[3] Model available at:
`https://huggingface.co/TechxGenus/Meta-Llama-3-70B-Instruct-GPTQ`



(a) (b) (c)

(d) (e) (f)

Figure 3: **ACES generates more diverse and more difficult problems.** *Diversity* (first row): semantic diversity (a), embedding diversity with the *codet5p* model (b) and the *deepseek-coder-1.3b* model (c). *Fitness* (second row): average fitness of valid puzzle generated over the last 160 generation attempts (d), QD-score (e) and distributions of fitness values over whole archives (f). ACES variants outperform baselines in terms of diversity, fitness and QD score (aggregated measure).

4.2 Quality of LLM-generated skill labels

We represent generated problems with skill descriptors generated by an LLM. This allows us to characterize abstract, semantic aspects of the generated problems that would be hard to capture with hand-written descriptor functions, and thus lets us optimize diversity in a space that is more aligned with human intuitive notions of variations in this programming domain. However, LLM labeling is stochastic and can be mis-aligned with human judgements. To validate the labeling process, we tagged 60 puzzles with semantic descriptors selected from the list of 20 (see Section 3.2) and, using them as ground truth, report a precision of 0.71, a recall of 0.75 and an F1 score of 0.73. Qualitatively, we found the labeler to be generally competent, always detecting the most salient skill descriptor and almost never labeling a puzzle with a totally irrelevant descriptor — although it did sometimes answer quite literally, calling a problem *set problem* when it used Python's *set* function. Appendix Figure 7 shows that the labeler uses almost all descriptors across the experiments, generally using 2-3 descriptor labels per puzzle.

4.3 ACES generates more diverse and challenging problems than existing approaches

More diverse The ability of an algorithm to generate diversity can be measured as the number of descriptor niches it manages to fill — stronger algorithms can reach more niches. On this metric, ACES and its variant ACES-ELM vastly outperform other baselines, reaching up to 700 different niches by the end of the experiment (Figure 3a). This shows that our approach can effectively generate a diversity of problems with respect to a target description space provided by the user: here the set of 20 programming skills defined in Section 3.2. Although ELM also leverages the same semantic descriptors, its undirected mutation operator does not seem to optimize diversity as effectively as the goal-directed generation operator of ACES variants.

However, the imperfection of the skill labeling could cast doubts on this result: could it be that ACES variants learn to generate problems that hack the labeler and force it to generate diverse labels that do not capture the true properties of the problems, this despite our preventive measures (see Section 3.3)? Furthermore, ELM-CVT and StaticGen do not leverage the semantic representation space, which may explain why they generate lower diversity in that space. To validate our results, we also measure *embedding diversity* metrics in a variety of other sentence embedding spaces not used at any point during training. This measure is obtained by embedding all generated puzzles with a given model, then computing the average cosine distance between all pairs of embeddings in the set. Figures 6a and 3c show these metrics for two standard embedding models: *codet5p-embedding* and *deepseek-coder-1.3b*: ACES and its variant significantly outperform others here too. ELM-CVT represents problems and optimizes for diversity in the *codet5p* embedding space, which explains its ability to generate higher diversity in that space than ELM (Figure 6a). However, this diversity does not transfer well to other diversity metrics (Figure 3a and Figure 3c), while the semantic diversity optimized by ACES variants does transfer well across diversity metrics (Figures 3a to 3c). Overall, these results show that by directly optimizing for semantic diversity through goal-directed exploration, ACES variants generate problems that are more semantically diverse, but also generally more diverse.

More challenging Figure 3d shows the evolution of the average fitness of the puzzles generated over the last 5 iterations (800 attempted puzzle generations). Here we see the ability of ACES variants to continuously generate increasingly challenging puzzles across the experiment, while ELM variants saturate earlier on. StaticGen, a static generation method, consistently generates puzzles with low fitness. Figure 3f shows the distribution of fitness of the whole archive at the end of experiments. These distributions are all bimodal, with problems having either very low fitness (solved most of the time) or very high fitness (solved only a couple times out of the 50 solving attempts). While StaticGen mostly generates simple problems (peak in the lowest fitness bin), all the algorithm optimizing for difficulty (all others) seem to do so efficiently — showing peaks in the highest fitness bin.

Finally, Figure 3e shows the evolution of *QD scores*, a metric used to evaluate QD methods that sums the fitness of the very best solution found in each filled niche. This metric captures both diversity (more niches means more terms in the sum) and local quality (higher quality solutions means higher terms in the sum). Here again, ACES variants significantly outperform other baselines. All these results are evidence that ACES variants efficiently produce a larger diversity of more challenging problems than existing algorithms.

Finetuning We conducted finetuning experiments using the Llama-3-8b model with datasets generated by WizardCoder (a variant of the state-of-the-art method WizardLM for generating synthetic data[Luo et al., 2023, Xu et al., 2023]), StaticGen (established baselines), and our proposed ACES-ELM method. We then evaluated the model's performance using the greedy pass@1 metric on a series of test sets. These test sets were equally composed of puzzles from our method and StaticGen with increasing difficulty levels, generated using a seed different from the training data.

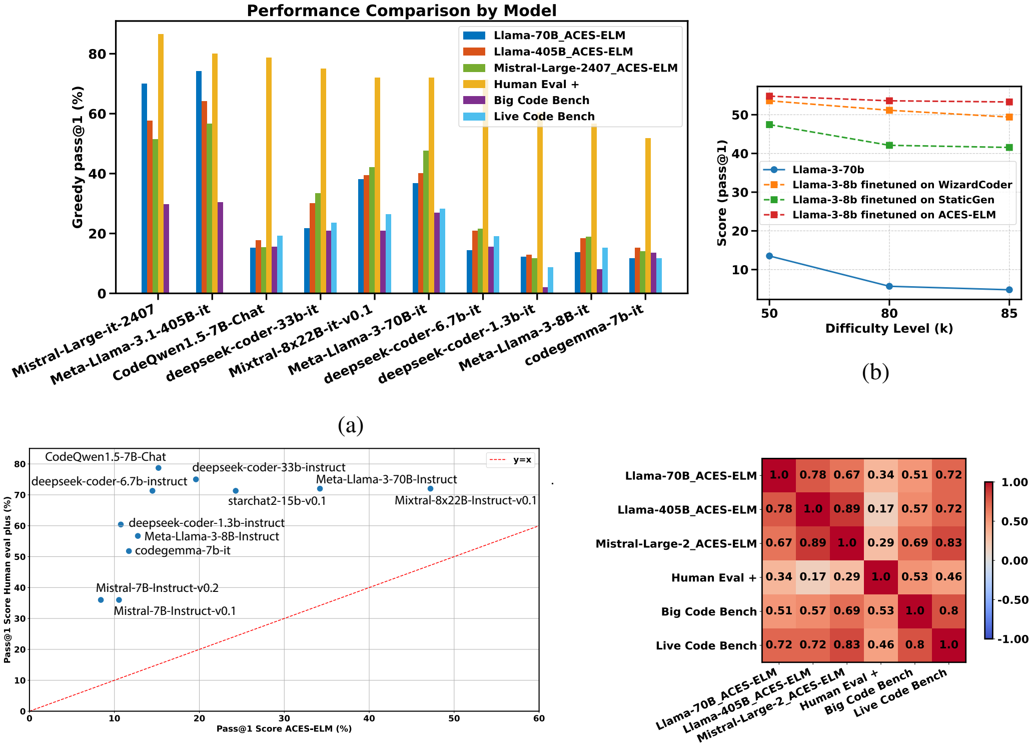
Results revealed that the Llama-3-8b model finetuned with ACES-ELM-generated data achieved a remarkable pass@1 score of 53.3 on the most challenging test set (see Figure 4b), significantly outperforming both baseline methods and the Llama-3-70B model. While models finetuned with WizardCoder and StaticGen showed improvements over the baseline (achieving pass@1 scores of 49.4 and 41.6, respectively), they consistently underperformed compared to the ACES-ELM-trained model. Notably, the performance gap between ACES-ELM and baseline methods widened as testset difficulty increased, demonstrating ACES-ELM's effectiveness in preparing models for complex coding challenges and generating high-quality training data.

4.4 ACES scales with LLM size

Diversity We found that ACES could generate more diverse and challenging problems, but what happened if we tried to use a larger LLM? We compare ACES archive generated using Llama-3-70b against larger models (Llama 405B model and Mistral large 2). Using those LLMs leads to a better Quality-Diversity score overall (see Figure 5), up to 25.6% for Mistral Large and 12.3% for Llama 405B (using Llama 405B and Mistral large 2 both for the difficulty metric and skill labeling). This demonstrates how ACES scales with models of larger size.

difficulty Evaluating Mistral Large and Llama 405B on each other's generated datasets reveals pass@1 scores of 56.7% and 58%, respectively (Figure 4a). These results demonstrate the effectiveness of our method in creating challenging benchmarks, even for state-of-the-art models, and highlight the transferability of difficulty measures across models with similar capabilities.

Furthermore, when tested on the archive generated by Llama-3-70B using ACES-ELM, Mistral Large 2 achieved a 70% pass@k score, while Llama-3-405B reached 74%. In contrast, Llama-3-70B itself attained a pass@1 score of 36.8%. These findings underscore the superior problem-solving



(a)

(b)

(c)

(d)

Figure 4: **(a)** Greedy pass@1 scores for various models on datasets generated by ACES-ELM with Llama-3-70b, Llama-3-405B, Mistral, on the HumanEval + benchmark, and on recent benchmark created to limit contamination and saturation compared to HumanEval (BigCodeBench and LiveCodeBench). Scores not found are represented as 0. **(b)** Greedy pass@1, after finetuning Llama-8b on archive generated by WizardCoder, StaticGen, and ACES-ELM, on a series of test sets equally composed of mixtures of puzzles from ACES-ELM and StaticGen. See rebuttal main text for additional details. **(c)** Pass@1 competence of all models on *HumanEval+* versus on problems generated by our best problem generator (ACES-ELM). Problems that are more challenging for a model are more generally challenging for others too (similar ranks in (a)). Models in the top left corner of (b) may be overfitting to HumanEval+, as their strong performance there do not translate into higher performance on our generated problems. **(d)** Correlation matrix of pass@1 scores of the different datasets. Our method achieves a correlation up to 0.83 with Live Code Bench and 0.69 with Big Code Bench, whereas HumanEval has only a correlation of 0.46 and 0.53 respectively.

capabilities of Mistral Large and Llama-405B. However, it's noteworthy that even these advanced models have not saturated the benchmark generated by Llama-3-70B, as there remains approximately 30% room for improvement to solve the benchmark fully.

4.5 ACES generates more challenging problems than the ones found in existing benchmarks

ACES variants generate problems that are more challenging than problems generated by other baselines. But how challenging are they really? Here, we measure the competence of 10 state-of-the-art LLM problem solvers and compare it to the performance of these same models on existing human-curated benchmarks. Figure 4a reports the pass@1 scores for 10 LLM solvers over two existing programming puzzles benchmarks: HumanEval + [Liu et al., 2024], Big Code Bench, Live Code Bench as well as over the set of problems generated by ACES variants (with Llama-3-70b, Llama-3.1-405b and Mistral Large 2) and other baselines in the experiments presented above. Results for additional models can be found in Appendix Figure 8. Overall, our generated sets are more challenging across models than the HumanEval sets usually used to benchmark code LLMs. *CodeQwen1.5-7B-Chat*, *Llama3-70B-instruct* and *Mixtral-8x22B-Instruct-v0.1*, one of the best open source models on HumanEval with 78.7, 72 and 72% pass@1 respectively, get a considerable drop in pass@1, falling to 15, 47 and 34% respectively on problems generated by ACES-ELM with Llama-3-70b (Figure 4c).

Our findings demonstrate the transferability of puzzle difficulty across various language models. Models exhibiting lower scores on our datasets generated with Llama 70B also have lower scores on the datasets generated by Llama 405B and Mistral, as well as on LiveCodeBench [Jain et al., 2024] (which is specifically designed to avoid contamination) and BigCodeBench [Zhuo et al., 2024]. Figure 4a illustrates this correlation, with HumanEval being the notable exception. This anomaly suggests potential contamination in models optimized for the HumanEval benchmark, particularly in smaller models. Figure 4d presents the correlation between pass@1 scores across all datasets, averaged over models. Our method has generated benchmarks that correlate more with uncontaminated benchmarks (LiveCodeBench and BigCodeBench) than HumanEval. Specifically, the Mistral-generated dataset using ACES-ELM achieved a correlation of 0.83 with LiveCodeBench. In contrast, HumanEval only reached a correlation 0.46 with Live Code Bench, underscoring the challenging nature and value of our generated benchmark for comparing LLMs.

5 Discussion

This paper presented ACES, an autotelic generative algorithm designed to generate a diversity of challenging Python programming puzzles. Our experiments showed that ACES generates problems that are both more diverse and more challenging than the ones generated by existing generative approaches or the ones found in human-curated benchmarks.

Limitations and improvements ACES has several limitations. The algorithm relies on several assumptions: 1) the labeling should be accurate, 2) the generator should be good at reaching its goals, 3) the correlation between the feature (descriptors and quality) of parent and children problems should be sufficiently high (heredity). Although these assumptions are only partially verified (see Section 4.2 and Appendix Figures 9 and 10), our results show that these are *good enough* for allowing the effective optimization of diversity and difficulty in our domain. We expect that progress along the three lines can be made by harnessing the increasing capabilities of future language models and will automatically translate into higher performing autotelic generative models of problems. The heredity property presents an interesting research question: in principle, it underlies all evolutionary methods making use of LLMs but, in practice, it has not been evaluated or discussed [Bradley et al., 2023a,b].

Creating and releasing challenging LLM benchmarks ACES allows for the automatic generation of programming puzzles tailored to LLM's current capabilities and we demonstrated that we generate puzzle sets that are more challenging than current code evaluations. This brings hope for generating tomorrow's next generation of code benchmarks, as today's evaluations are almost saturated. However, important steps need to be performed before this can be done. First, as is done in Schuster et al. [2021], we currently only test solutions with one set of arguments, there should be tests with a wide range to measure solution robustness. Empirical difficulty is not the only thing to be expected of exercises: the puzzles need to make sense and be hard for the good reasons. This requires a more complete picture of a puzzle's quality than difficulty alone, which remains an open problem. Overall, we believe that open-ended algorithms will play an increasing role in automatically evaluating LLMs in the future [Samvelyan et al., 2024].

Other applications ACES is a general algorithm that can be easily translated to other application domains by letting the user define a new descriptor and quality function. Swapping the difficulty metric with a more subjective objective estimated by an LLM could let ACES generate problems adapted for human students of a specific level in educational contexts, for example. In artificial intelligence, one could envision a self-play loop where a learning agent iteratively generates a diversity of problems maximizing its current learning progress (quality metric), then trains on them to augment its competence [Sukhbaatar et al., 2017, Silver et al., 2017, Colas et al., 2022].

Broader impacts Open-ended exploration algorithms in general have wide-ranging implications when scaled up. They could potentially help in discovering harmful artifacts (in code domains, harmful bots or cyberattack programs), as well as help find solutions or red-team existing systems. We note that an evolutionary optimization algorithm such as ACES needs extensive fitness feedback, either limiting its use by bad actors or making them detectable. Positive applications of open-ended exploration algorithms applied to problem generation range from educational technologies to automated scientific discovery, helping shoulder some of the rising needs of the future.

References

Junyi Chu and Laura E. Schulz. Play, curiosity, and cognition. *Annual Review of Developmental Psychology*, 2(1):317–343, 2020. doi: 10.1146/annurev-devpsych-070120-014806. URL `https://doi.org/10.1146/annurev-devpsych-070120-014806`.

Gaia Molinaro and Anne GE Collins. A goal-centric outlook on learning. *Trends in Cognitive Sciences*, 2023.

Benjamin A Burton and Mathias Hiron. Creating informatics olympiad tasks: exploring the black art. *Olympiads in Informatics*, 2:16–36, 2008.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating Large Language Models Trained on Code, July 2021. URL `http://arxiv.org/abs/2107.03374`. arXiv:2107.03374 [cs].

Misha Gromov. *Great circle of mysteries*. Birkhauser, Basel, Switzerland, 1 edition, May 2018.

Junyi Chu, Joshua B. Tenenbaum, and Laura E. Schulz. In praise of folly: flexible goals and human cognition. *Trends in Cognitive Sciences*, 2024/05/20 2024. ISSN 1364-6613. doi: 10.1016/j.tics.2024.03.006. URL `https://doi.org/10.1016/j.tics.2024.03.006`.

Jürgen Schmidhuber. Powerplay: Training an increasingly general problem solver by continually searching for the simplest still unsolvable problem. *Frontiers in psychology*, 4:313, 2013.

Vincent Herrmann, Louis Kirsch, and Jürgen Schmidhuber. Learning one abstract bit at a time through self-invented experiments encoded as neural networks. *arXiv preprint arXiv:2212.14374*, 2022.

Cédric Colas, Tristan Karch, Olivier Sigaud, and Pierre-Yves Oudeyer. Autotelic Agents with Intrinsically Motivated Goal-Conditioned Reinforcement Learning: a Short Survey, July 2022. URL `http://arxiv.org/abs/2012.09830`. arXiv:2012.09830 [cs].

Rémy Portelas, Cédric Colas, Lilian Weng, Katja Hofmann, and Pierre-Yves Oudeyer. Automatic curriculum learning for deep rl: A short survey. *arXiv preprint arXiv:2003.04664*, 2020.

Jonathan Grizou, Laurie J. Points, Abhishek Sharma, and Leroy Cronin. A curious formulation robot enables the discovery of a novel protocell behavior. *Science Advances*, 6(5):eaay4237, 2020. doi: 10.1126/sciadv.aay4237. URL `https://www.science.org/doi/abs/10.1126/sciadv.aay4237`.

Mayalen Etcheverry. *Curiosity-driven AI for Science: Automated Discovery of Self-Organized Structures*. PhD thesis, Inria & Labri, Université Bordeaux, 2023.

Tal Schuster, Ashwin Kalyan, Alex Polozov, and Adam Tauman Kalai. Programming Puzzles. June 2021. URL `https://openreview.net/forum?id=fe_hCc4RBrg`.

Patrick Haluptzok, Matthew Bowers, and Adam Tauman Kalai. Language Models Can Teach Themselves to Program Better. In *The Eleventh International Conference on Learning Representations*, 2022.

Or Honovich, Thomas Scialom, Omer Levy, and Timo Schick. Unnatural instructions: Tuning language models with (almost) no human labor. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 14409–14428, Toronto, Canada, July 2023. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.806. URL https://aclanthology.org/2023.acl-long.806.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, et al. Textbooks are all you need. *arXiv preprint arXiv:2306.11644*, 2023.

Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites. *arXiv preprint arXiv:1504.04909*, 2015.

Justin Pugh, Lisa Soros, and Kenneth Stanley. Quality Diversity: A New Frontier for Evolutionary Computation. *Frontiers in Robotics and AI*, 3, July 2016. doi: 10.3389/frobt.2016.00040.

Herbie Bradley, Honglu Fan, Francisco Carvalho, Matthew Fisher, Louis Castricato, reciprocated, dmayhem93, Shivanshu Purohit, and Joel Lehman. OpenELM, January 2023a. URL https://github.com/CarperAI/OpenELM.

Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

Joel Lehman and Kenneth O Stanley. Abandoning objectives: evolution through the search for novelty alone. *Evol. Comput.*, 19(2):189–223, February 2011a.

Joel Lehman and Kenneth O Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, pages 211–218, 2011b.

Antoine Cully and Yiannis Demiris. Quality and diversity optimization: A unifying modular framework. *IEEE Transactions on Evolutionary Computation*, 22(2):245–259, 2018a. doi: 10.1109/TEVC.2017.2704781.

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through Large Models, June 2022. URL http://arxiv.org/abs/2206.08896. arXiv:2206.08896 [cs].

Adrien Baranes and Pierre-Yves Oudeyer. Active learning of inverse models with intrinsically motivated goal exploration in robots. *Robotics and Autonomous Systems*, 61(1):49–73, January 2013. ISSN 0921-8890. doi: 10.1016/j.robot.2012.05.008. URL https://www.sciencedirect.com/science/article/pii/S0921889012000644.

Mayalen Etcheverry, Clément Moulin-Frier, and Pierre-Yves Oudeyer. Hierarchically organized latent modules for exploratory search in morphogenetic systems. *Advances in Neural Information Processing Systems*, 33:4846–4859, 2020.

Sébastien Forestier, Rémy Portelas, Yoan Mollard, and Pierre-Yves Oudeyer. Intrinsically motivated goal exploration processes with automatic curriculum learning. *The Journal of Machine Learning Research*, 23(1):6818–6858, 2022. Publisher: JMLRORG.

Joel Lehman and Kenneth O. Stanley. Evolving a diversity of virtual creatures through novelty search and local competition. In *Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation*, GECCO '11, page 211–218, New York, NY, USA, 2011c. Association for Computing Machinery. ISBN 9781450305570. doi: 10.1145/2001576.2001606. URL https://doi.org/10.1145/2001576.2001606.

Clément Moulin-Frier, Sao M Nguyen, and Pierre-Yves Oudeyer. Self-organization of early vocal development in infants and machines: the role of intrinsic motivation. *Frontiers in psychology*, 4:1006, 2014.

Pierre-Yves Oudeyer and Linda B Smith. How evolution may work through curiosity-driven developmental process. *Topics in Cognitive Science*, 8(2):492–502, 2016.

Chris Reinke, Mayalen Etcheverry, and Pierre-Yves Oudeyer. Intrinsically motivated discovery of diverse patterns in self-organizing systems. *arXiv preprint arXiv:1908.06663*, 2019.

Elliot Meyerson, Mark J Nelson, Herbie Bradley, Arash Moradi, Amy K Hoover, and Joel Lehman. Language model crossover: Variation through few-shot prompting. *arXiv preprint arXiv:2302.12170*, 2023.

Angelica Chen, David M Dohan, and David R So. Evoprompting: Language models for code-level neural architecture search. *arXiv preprint arXiv:2302.14838*, 2023.

Muhammad U Nasir, Sam Earle, Julian Togelius, Steven James, and Christopher Cleghorn. Llmatic: Neural architecture search via large language models and quality-diversity optimization. *arXiv preprint arXiv:2306.01102*, 2023.

Yuntao Bai, Saurav Kadavath, Sandipan Kundu, Amanda Askell, Jackson Kernion, Andy Jones, Anna Chen, Anna Goldie, Azalia Mirhoseini, Cameron McKinnon, Carol Chen, Catherine Olsson, Christopher Olah, Danny Hernandez, Dawn Drain, Deep Ganguli, Dustin Li, Eli Tran-Johnson, Ethan Perez, Jamie Kerr, Jared Mueller, Jeffrey Ladish, Joshua Landau, Kamal Ndousse, Kamile Lukosuite, Liane Lovitt, Michael Sellitto, Nelson Elhage, Nicholas Schiefer, Noemi Mercado, Nova DasSarma, Robert Lasenby, Robin Larson, Sam Ringer, Scott Johnston, Shauna Kravec, Sheer El Showk, Stanislav Fort, Tamera Lanham, Timothy Telleen-Lawton, Tom Conerly, Tom Henighan, Tristan Hume, Samuel R. Bowman, Zac Hatfield-Dodds, Ben Mann, Dario Amodei, Nicholas Joseph, Sam McCandlish, Tom Brown, and Jared Kaplan. Constitutional AI: Harmlessness from AI Feedback, December 2022. URL `http://arxiv.org/abs/2212.08073`. arXiv:2212.08073 [cs].

Harrison Lee, Samrat Phatale, Hassan Mansoor, Kellie Lu, Thomas Mesnard, Colton Bishop, Victor Carbune, and Abhinav Rastogi. Rlaif: Scaling reinforcement learning from human feedback with ai feedback. *arXiv preprint arXiv:2309.00267*, 2023.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

Jenny Zhang, Joel Lehman, Kenneth Stanley, and Jeff Clune. OMNI: Open-endedness via Models of human Notions of Interestingness, June 2023. URL `http://arxiv.org/abs/2306.01711`. arXiv:2306.01711 [cs].

Martin Klissarov, Pierluca D'Oro, Shagun Sodhani, Roberta Raileanu, Pierre-Luc Bacon, Pascal Vincent, Amy Zhang, and Mikael Henaff. Motif: Intrinsic motivation from artificial intelligence feedback. *arXiv preprint arXiv:2310.00166*, 2023.

Noveen Sachdeva, Benjamin Coleman, Wang-Cheng Kang, Jianmo Ni, Lichan Hong, Ed H Chi, James Caverlee, Julian McAuley, and Derek Zhiyuan Cheng. How to train data-efficient llms. *arXiv preprint arXiv:2402.09668*, 2024.

Herbie Bradley, Andrew Dai, Hannah Teufel, Jenny Zhang, Koen Oostermeijer, Marco Bellagente, Jeff Clune, Kenneth Stanley, Grégory Schott, and Joel Lehman. Quality-diversity through ai feedback. *arXiv preprint arXiv:2310.13032*, 2023b.

Mikayel Samvelyan, Sharath Chandra Raparthy, Andrei Lupu, Eric Hambro, Aram H Markosyan, Manish Bhatt, Yuning Mao, Minqi Jiang, Jack Parker-Holder, Jakob Foerster, et al. Rainbow teaming: Open-ended generation of diverse adversarial prompts. *arXiv preprint arXiv:2402.16822*, 2024.

Cédric Colas, Laetitia Teodorescu, Pierre-Yves Oudeyer, Xingdi Yuan, and Marc-Alexandre Côté. Augmenting Autotelic Agents with Large Language Models. *arXiv preprint arXiv:2305.12487*, 2023.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. Voyager: An Open-Ended Embodied Agent with Large Language Models, May 2023a. URL `http://arxiv.org/abs/2305.16291`. arXiv:2305.16291 [cs].

Yuqing Du, Olivia Watkins, Zihan Wang, Cédric Colas, Trevor Darrell, Pieter Abbeel, Abhishek Gupta, and Jacob Andreas. Guiding pretraining in reinforcement learning with large language models. *arXiv preprint arXiv:2302.06692*, 2023.

Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Yizhong Wang, Yeganeh Kordi, Swaroop Mishra, Alisa Liu, Noah A. Smith, Daniel Khashabi, and Hannaneh Hajishirzi. Self-instruct: Aligning language models with self-generated instructions. In Anna Rogers, Jordan Boyd-Graber, and Naoaki Okazaki, editors, *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 13484–13508, Toronto, Canada, July 2023b. Association for Computational Linguistics. doi: 10.18653/v1/2023.acl-long.754. URL `https://aclanthology.org/2023.acl-long.754`.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

Marah Abdin, Sam Ade Jacobs, Ammar Ahmad Awan, Jyoti Aneja, Ahmed Awadallah, Hany Awadalla, Nguyen Bach, Amit Bahree, Arash Bakhtiari, Harkirat Behl, et al. Phi-3 technical report: A highly capable language model locally on your phone. *arXiv preprint arXiv:2404.14219*, 2024.

Can Xu, Qingfeng Sun, Kai Zheng, Xiubo Geng, Pu Zhao, Jiazhan Feng, Chongyang Tao, and Daxin Jiang. Wizardlm: Empowering large language models to follow complex instructions. *arXiv preprint arXiv:2304.12244*, 2023.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Dingli Yu, Simran Kaur, Arushi Gupta, Jonah Brown-Cohen, Anirudh Goyal, and Sanjeev Arora. Skill-mix: A flexible and expandable family of evaluations for ai models. *arXiv preprint arXiv:2310.17567*, 2023.

Vassilis Vassiliades, Konstantinos Chatzilygeroudis, and Jean-Baptiste Mouret. Using Centroidal Voronoi Tessellations to Scale Up the Multi-dimensional Archive of Phenotypic Elites Algorithm, July 2017. URL `http://arxiv.org/abs/1610.05729`. arXiv:1610.05729 [cs].

Qiang Du, Vance Faber, and Max Gunzburger. Centroidal Voronoi Tessellations: Applications and Algorithms. *SIAM Review*, 41(4):637–676, 1999. URL `http://www.jstor.org/stable/2653198`.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

Naman Jain, King Han, Alex Gu, Wen-Ding Li, Fanjia Yan, Tianjun Zhang, Sida Wang, Armando Solar-Lezama, Koushik Sen, and Ion Stoica. Livecodebench: Holistic and contamination free evaluation of large language models for code. *arXiv preprint arXiv:2403.07974*, 2024.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, et al. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *arXiv preprint arXiv:2406.15877*, 2024.

Sainbayar Sukhbaatar, Zeming Lin, Ilya Kostrikov, Gabriel Synnaeve, Arthur Szlam, and Rob Fergus. Intrinsic motivation and automatic curricula via asymmetric self-play. *arXiv preprint arXiv:1703.05407*, 2017.

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

Ashvin V Nair, Vitchyr Pong, Murtaza Dalal, Shikhar Bahl, Steven Lin, and Sergey Levine. Visual reinforcement learning with imagined goals. *Advances in neural information processing systems*, 31, 2018.

Adrien Laversanne-Finot, Alexandre Pere, and Pierre-Yves Oudeyer. Curiosity driven exploration of learned disentangled goal spaces. In *Conference on Robot Learning*, pages 487–504. PMLR, 2018.

Chris Reinke, Mayalen Etcheverry, and Pierre-Yves Oudeyer. Intrinsically Motivated Discovery of Diverse Patterns in Self-Organizing Systems. In *International Conference on Learning Representations (ICLR)*, 2020.

Antoine Cully and Yiannis Demiris. Hierarchical behavioral repertoires with unsupervised descriptors. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 69–76, 2018b.

A Appendix

A.1 Additional Related Work

Descriptor spaces for exploration methods In all open-ended exploration methods, one must define a Behavioral Characterization (BC) space to characterize novelty. The earliest works used predefined low-dimensional descriptors to represent generated artefacts [Lehman and Stanley, 2011a, Baranes and Oudeyer, 2013, Mouret and Clune, 2015], which constrains the search along a handful of features one can code a descriptor for. More recent works have relied on higher-dimensional learned or pretrained embedding functions [Nair et al., 2018, Laversanne-Finot et al., 2018, Reinke et al., 2020], and even hierarchies of such spaces, each representing different perceptual features of the generated artefacts [Cully and Demiris, 2018b, Etcheverry et al., 2020]. Diversity-search algorithms sometimes need to be adapted to work with such high-dimensional spaces whose discretization leads to an exponential number of cells [Vassiliades et al., 2017]. But the main issue is that they are hardly interpretable and might not always align with the dimensions of variation humans find meaningful. With ACES, we propose an autotelic diversity-producing algorithm that constrains the search along a set of abstract, interpretable and hard-to-compute features of interest evaluated by LLMs.

A.2 Prompts

Here are the various prompts we use for ACES and all baselines.

Skills description. Skills description used to label problem. see prompt A.2

List of semantic descriptors

0 - **String Manipulation**
1 - **Mathematical Operations**
2 - **Conditional Logic**
3 - **Recursion**
4 - **Brute Force Search**
5 - **Dynamic Programming**
6 - **Greedy Algorithms**
7 - **Backtracking**
8 - **Set Operations**
9 - **Permutations and Combinations**
10 - **Probability and Statistics**
11 - **Pattern Recognition**
12 - **Sorting and Ordering**
13 - **Binary Operations (bitwise shifting, AND, OR)**
14 - **Geometry and Coordinate Manipulation**
15 - **Algorithm Optimization**
16 - **Number Theory (factors, primes, etc.)**
17 - **Graph Theory (paths, edges, vertices)**
18 - **Array Indexing**
19 - **Hashing**

Example of puzzle labelling

Puzzle to label:

```
def f(s: str) -> bool:\\
    """Find a string with 1000 'o's but no two adjacent 'o's.
       """\\
    return s.count('o') == 1000 and s.count('oo') == 0\\
def g():\\
    return ('h' + 'o') * 1000\\
assert f(g())\\
```

Assistant: To solve the problem, we need to understand the logic of the ‘f‘ function and how it checks for the presence of 1000 ’o’s and no adjacent ’o’s. We also need to understand the implementation of the ‘g‘ function, which generates the string with 1000 ’o’s.

Based on this understanding, the necessary programming skills are:
- 1: Counting and combinatorics (to understand the counting of ’o’s and ’oo’s in the string)
- 5: String Manipulation (to understand the string concatenation and searching)

Therefore, the list of indices for the problem is: [1, 5]

Prompt for the puzzle generator of ACES. This prompt is used for ACES and all autotelic variants.

ACES

User:
Consider Python Programming Puzzles (P3). P3 consists of two functions: a problem function ‘f‘ and its corresponding solution ‘g‘. The challenge lies in constructing a SAT problem ‘f‘ and a function ‘g‘ such that ‘f(g())‘ evaluates to ‘True‘

Main Rules:
- Each puzzle includes two functions: ‘def f(...)‘ and ‘def g(...)‘.
- The first argument of ‘f‘ is always the output from ‘g()‘.
- Ensure ‘f‘ and ‘g‘ have matching argument signatures (e.g., ‘def f(solution, arg1=value1, arg2=value2, ...)‘ and ‘def g(arg1=value1, arg2=value2, ...)‘). You also need to set the value of argument of f (arg1,arg2,...) and g when you define them.
- Avoid using ‘f‘ inside ‘g‘, and ‘g‘ inside ‘f‘.
- Include any necessary imports so your code runs smoothly.
- Give a clear Puzzle description that must be brief and diverse compared to the other puzzles.
- Make sure the puzzle is self-contained within these two functions.
- Make sure that each puzzle have just all required skills (see below)

P3 Format:
Puzzle description: A two to four sentence summary of the puzzle’s content. To explain what is the problem ‘f‘, and how you can solve it with ‘g‘.

~~~
```python
def f(solution, args=...) -> bool:
    # Python code to test the solution returned by g.
    # This function is a test unit and must return True if the
        solution is correct, and False otherwise.

def g(args=...) -> solution:
    # Python code to generate a solution for the problem.
    # The solution should generalize to all possible args.
    return solution

assert f(g()) == True
```
~~~

Examples:

Puzzle 0:
Puzzle description: [puzzle description]

- Difficulty score: [puzzle score] out of 100

- This puzzle has the following skills:
[skillslist]

[Python Programming Puzzle]

Puzzle 1:
...
Puzzle 2:
...

Generate 5 P3 similar to previous Examples. Ensure that all new puzzles are more challenging than Puzzle from previous examples.
You should aim to generate puzzles with a Difficulty score between 90 and 100 out of 100.

Please make sure that new puzzles have JUST ALL the following skills:
[list target skills]

New 5 problems:
Assistant:

ACES-ELM

User:
Consider Python Programming Puzzles (P3). P3 consists of two functions: a problem function ‘f‘ and its corresponding solution ‘g‘. The challenge lies in constructing a SAT problem ‘f‘ and a function ‘g‘ such that ‘f(g())‘ evaluates to ‘True‘

Main Rules:
- Each puzzle includes two functions: ‘def f(...)‘ and ‘def g(...)‘.
- The first argument of ‘f‘ is always the output from ‘g()‘.
- Ensure ‘f‘ and ‘g‘ have matching argument signatures (e.g., ‘def f(solution, arg1=value1, arg2=value2, ...)‘ and ‘def g(arg1=value1, arg2=value2, ...)‘). You also need to set the value of argument of f (arg1,arg2,...) and g when you define them.
- Avoid using ‘f‘ inside ‘g‘, and ‘g‘ inside ‘f‘.
- Include any necessary imports so your code runs smoothly.
- Give a clear Puzzle description that must be brief and diverse compared to the other puzzles.
- Make sure the puzzle is self-contained within these two functions.

- Make sure that each puzzle have just all required skills (see below)

P3 Format:
Puzzle description: A two to four sentence summary of the puzzle’s content. To explain what is the problem ‘f‘, and how you can solve it with ‘g‘.

~~~
```python
def f(solution, args=...) -> bool:
    # Python code to test the solution returned by g.
    # This function is a test unit and must return True if the
        solution is correct, and False otherwise.

def g(args=...) -> solution:
    # Python code to generate a solution for the problem.
    # The solution should generalize to all possible args.
    return solution

assert f(g()) == True
```
~~~

Examples:

Puzzle 0:
Puzzle description: [puzzle description]

- Difficulty score: [puzzle score] out of 100

- This puzzle has the following skills:
[skills list]

[Python Programming Puzzle]

Puzzle 1:
...
Puzzle 2:
...

Generate 5 P3 similar to the last Examples (Puzzle 2). Ensure that all new puzzles are more challenging than Puzzle 2.
You should aim to generate puzzles with a Difficulty score between 90 and 100 out of 100.

Please make sure that new puzzles have JUST ALL the following skills:
[list target skills]

New 5 problems inspired by Puzzle 2:
Assistant:

Prompt for the puzzle generator of Static gen.

Static gen

User: Consider Python Programming Puzzles (P3). P3 consists of two functions: a problem function ‘f‘ and its corresponding solution ‘g‘. The challenge lies in constructing a SAT problem ‘f‘ and a function ‘g‘ such that ‘f(g())‘ evaluates to ‘True‘

Main Rules:
- Each puzzle includes two functions: ‘def f(...)‘ and ‘def g(...)‘.
- The first argument of ‘f‘ is always the output from ‘g()‘.

- Ensure ‘f‘ and ‘g‘ have matching argument signatures (e.g., ‘def f(solution, arg1=value1, arg2=value2, ...)‘ and ‘def g(arg1=value1, arg2=value2, ...)‘). You also need to set the value of argument of f (arg1,arg2,...) and g when you define them.
- Avoid using ‘f‘ inside ‘g‘, and ‘g‘ inside ‘f‘.
- Include any necessary imports so your code runs smoothly.
- Give a clear Puzzle description that must be brief and diverse compared to the other puzzles.
- Make sure the puzzle is self-contained within these two functions.

P3 Format:
Puzzle description: A two to four sentence summary of the puzzle’s content. To explain what is the problem ‘f‘, and how you can solve it with ‘g‘.

~~~
```python
def f(solution, args=...) -> bool:
    # Python code to test the solution returned by g.
    # This function is a test unit and must return True if the
        solution is correct, and False otherwise.

def g(args=...) -> solution:
    # Python code to generate a solution for the problem.
    # The solution should generalize to all possible args.
    return solution

assert f(g()) == True
```
~~~

Examples:

Puzzle 0:
Puzzle description: [puzzle description]

- Difficulty score: [puzzle score] out of 100

- This puzzle has the following skills:
[skillslist]

[Python Programming Puzzle]

Puzzle 1:
...
Puzzle 2:
...

Generate 5 different P3 similar to previous Examples.

New 5 problems:
[list target skills]

New 5 problems inspired by Puzzle 2
Assistant:

Prompt for the puzzle generator of ELM and ELM semantic. This prompt is used for non-autotelic baselines.

ELM and ELM semantic

User: Consider Python Programming Puzzles (P3). P3 consists of two functions: a problem function ‘f‘ and its corresponding solution ‘g‘. The challenge lies in constructing a SAT problem ‘f‘ and a function ‘g‘ such that ‘f(g())‘ evaluates to ‘True‘

Main Rules:
- Each puzzle includes two functions: ‘def f(...)‘ and ‘def g(...)‘.
- The first argument of ‘f‘ is always the output from ‘g()‘.
- Ensure ‘f‘ and ‘g‘ have matching argument signatures (e.g., ‘def f(solution, arg1=value1, arg2=value2, ...)‘ and ‘def g(arg1=value1, arg2=value2, ...)‘). You also need to set the value of argument of f (arg1,arg2,...) and g when you define them.
- Avoid using ‘f‘ inside ‘g‘, and ‘g‘ inside ‘f‘.
- Include any necessary imports so your code runs smoothly.
- Give a clear Puzzle description that must be brief and diverse compared to the other puzzles.
- Make sure the puzzle is self-contained within these two functions.

P3 Format:
Puzzle description: A two to four sentence summary of the puzzle’s content. To explain what is the problem ‘f‘, and how you can solve it with ‘g‘.

~~~
```python
def f(solution, args=...) -> bool:
    # Python code to test the solution returned by g.
    # This function is a test unit and must return True if the
        solution is correct, and False otherwise.

def g(args=...) -> solution:
    # Python code to generate a solution for the problem.
    # The solution should generalize to all possible args.
    return solution

assert f(g()) == True
```
~~~

Examples:

Puzzle 0:
Puzzle description: [puzzle description]

- Difficulty score: [puzzle score] out of 100

- This puzzle has the following skills:
[skillslist]

[Python Programming Puzzle]

Puzzle 1:
...
Puzzle 2 (to mutate):
...

Generate 5 P3 similar to the last Examples (Puzzle 2). Ensure that all new puzzles are more challenging than Puzzle 2.
You should aim to generate puzzles with a Difficulty score between 90 and 100 out of 100.

Please make sure that new puzzles have JUST ALL the following skills:
[list target skills]

New 5 problems inspired by Puzzle 2
Assistant:

Description prompt

User: A Python programming puzzle is defined by two functions, the problem f(solution, arg1=value1, arg2=value2, ..) and the solution. f defines an algorithmic puzzle, and the solution solves this puzzle.
You should pay a particular attention that the puzzle is solved if and only if **f(solution) == True**.
Your role is to write a one or two sentence the description of the puzzle's goal (what the solution should be), remember that the solution that satisfy the goal must be given as the first argument of ‘f‘.
You can start by: ’Find the solution: "arg solution" (describe its type shortly) that should (here you should speak about the solution: "arg solution" and how it should solve all the constraints of the puzzle with respect to others args (describe their types shortly)) ...’.
For example:
’Given a string ‘str1‘, find the length of the longest substring without repeating characters.’
’Given two sorted arrays ‘nums1‘ and ‘nums2‘ of size ‘m‘ and ‘n‘ respectively, return the median of the two sorted arrays.’

The puzzle is:
...

Assistant:

Skill labeling prompt

User: You are a helpful assistant to a Professor teaching a programming course in Python. The Professor want to give Pyhton programming puzzles to his Computer Science student to teach them Python.
A Python programming puzzle is defined by two functions, the puzzle f(. . .) and the solution g(. . .). f defines an algorithmic challenge, and g solves this challenge. g is a solution to f if and only if f(g()) == True.
The Professor want to evaluate the diversity of those puzzles, can you label the following puzzle given the following list of topics, please?
The list of topics is:

skills list

The puzzle is:
...

Respond with two or three sentence explaining the topics used in the puzzle.
Then summarize your response by giving a list from 1 to 5 index corresponding to topics that are actually used in the puzzle above in this format: ’The list of skill use is: [].’ where [] is the list of index of the topics used in the puzzle for example [3,5,6].

Assistant:

Solver prompt

User: You will be given a function. Respond only in code with a correct, efficient implementation of the function. You will need to generate the correct solutions (g), for the Problem 2 that satisfies the condition f(g()) == True.

Problem 0:
def f(...

Solution 0:
def g(...
Problem 1:
def f(...
Solution 1:
def g(...

Now you need to give the solution (def g(...)) to the following Problem 2 that satisfies the condition $f(g()) == True$.

Problem 2:
def f(...

Assistant:

Example Generation label

Puzzle to mutate:

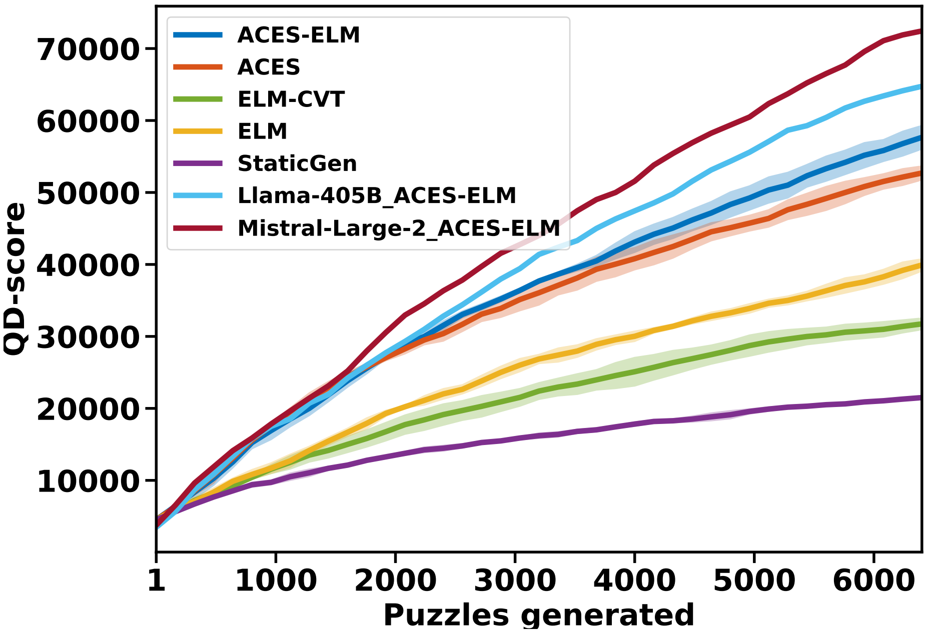
```
from typing import*
def f(n: int, lst=['apple', 'banana', 'orange', 'grape']) ->
    bool:
    """Check if the given element n is a prefix of any element
         in the list lst"""
    for word in lst:
        if word.startswith(n):
            return True
    return False

def g(lst=['apple', 'banana', 'orange', 'grape']):
    return lst[1]

assert f(g())
```

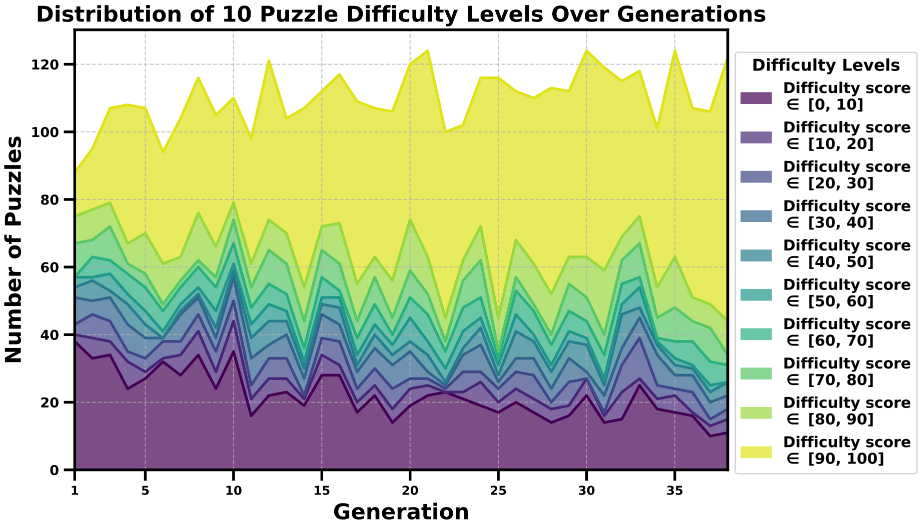
Assistant:

```
from typing import*
import math
def f(n: int, lst=[1, 2, 3, 4, 5]) -> bool:
    """Check if the given list contains any perfect square
        number and if it is divisible by n."""
    for num in lst:
        if math.isqrt(num)**2 == num:
            return n % num == 0
    return False
def g(lst=[1, 2, 3, 4, 5]):
    for num in lst:
        if math.isqrt(num)**2 == num:
            return num
    return None
assert f(g())
```



(a)

Figure 5: **(b)** Quality-Diversity (QD) score updated figure with ACES-ELM using state-of-the-art open weight LLM (Llama-3-405B and Mistral Large 2)



(a)

Figure 6: **(e)** Evolution of Puzzle Difficulty Distribution generated by ACES: Decile Analysis of Generated Puzzles over generations.

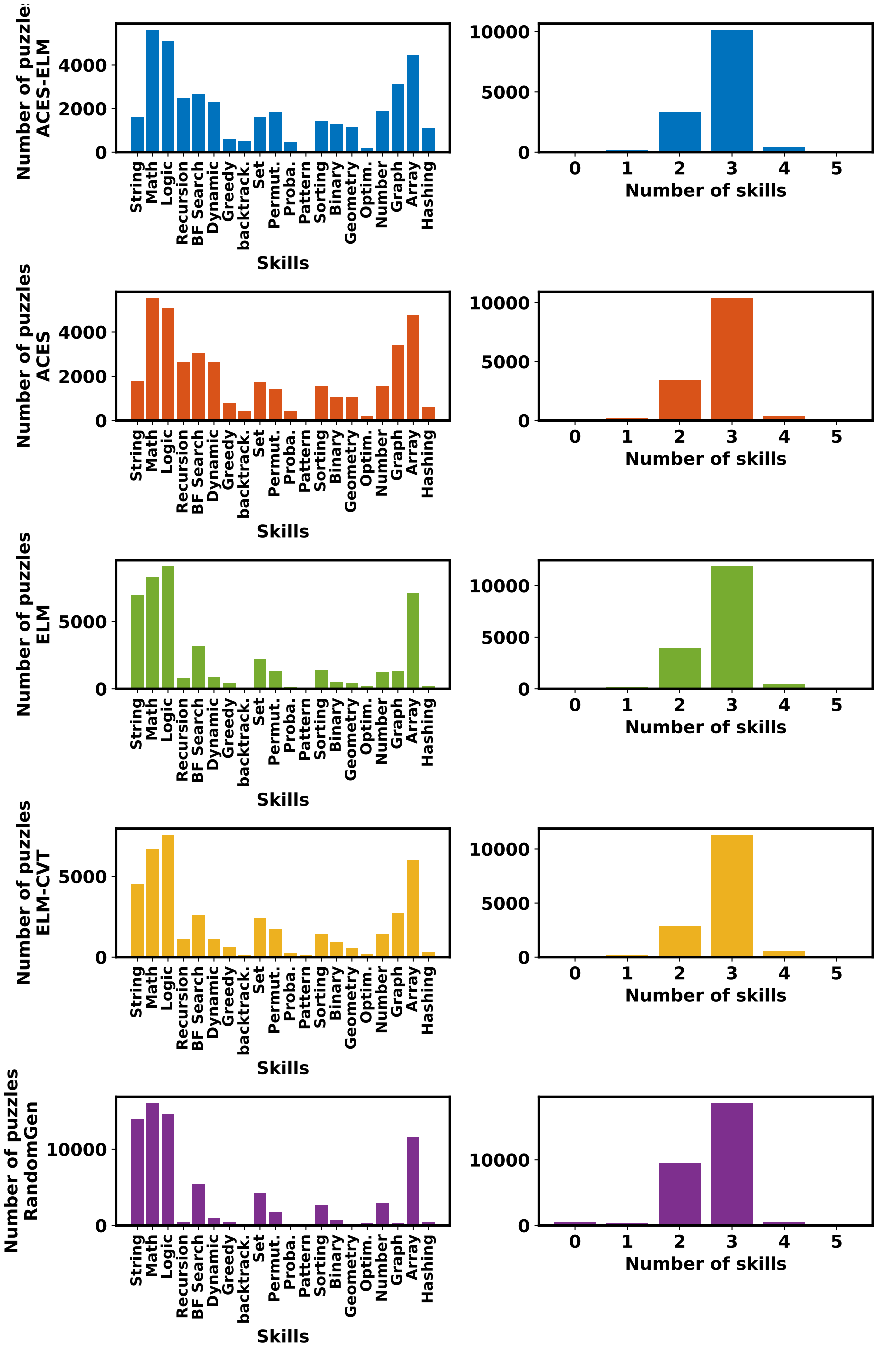


Figure 7: Distribution of labeled skills (left) and number of skills labeled (right) for all algorithms (rows).

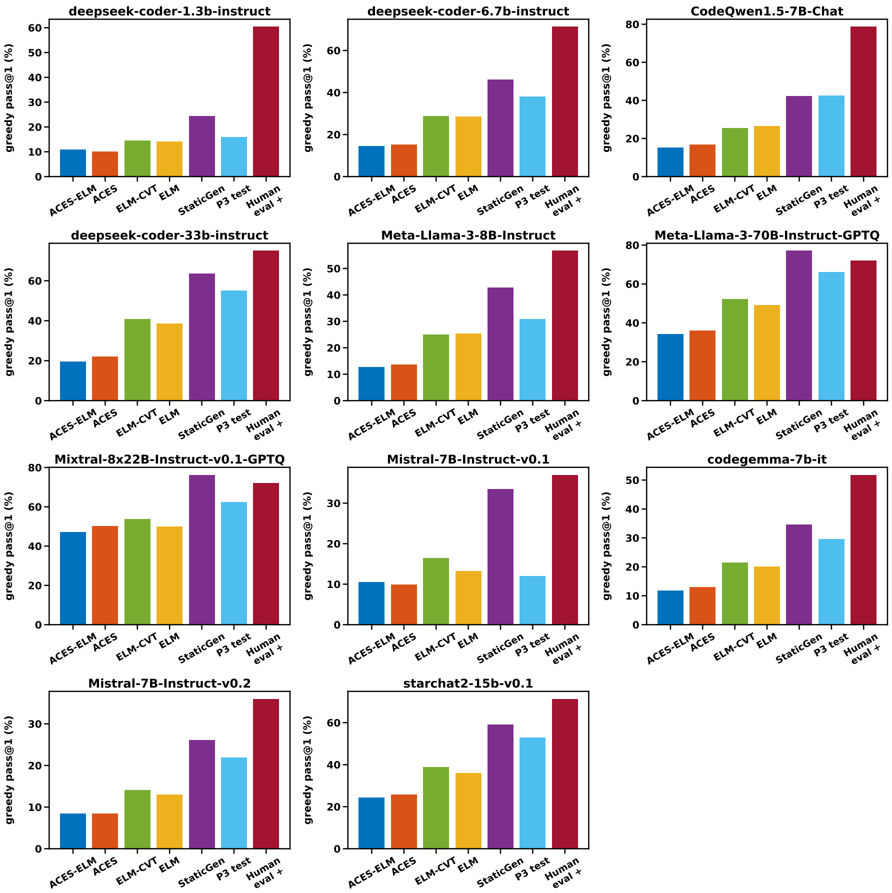


Figure 8: Pass@1 competence of various state-of-the-art problem solvers on existing benchmarks (*HumanEval+* and *P3 test*) and on problems generated by our algorithms.

A.3 Semantic descriptor diversity

A.4 Performance Comparison

A.5 Heridity

A.6 Examples of generated puzzles

In this section, we present a few puzzles and solutions generated by our different methods and examine them qualitatively.

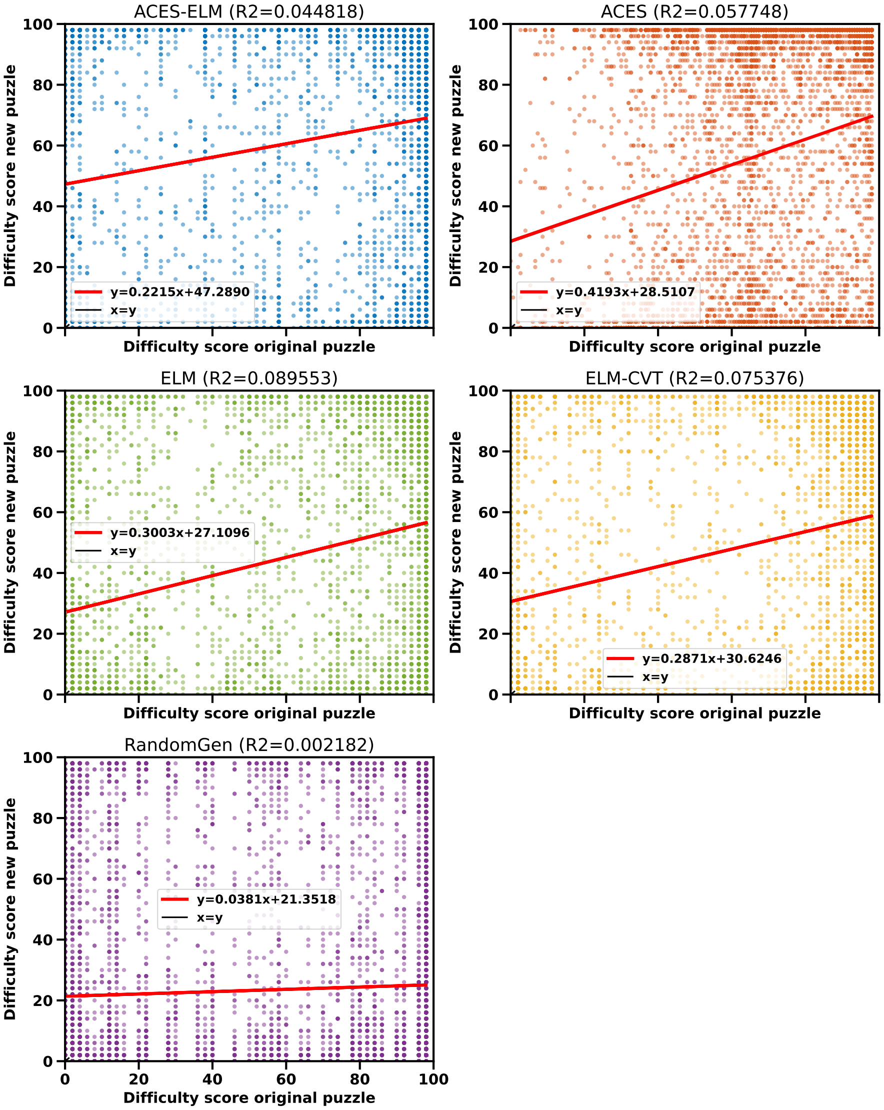


Figure 9: Evolution of the difficulty score from puzzles used as a few-shot example to generated puzzles based on those previous puzzles.

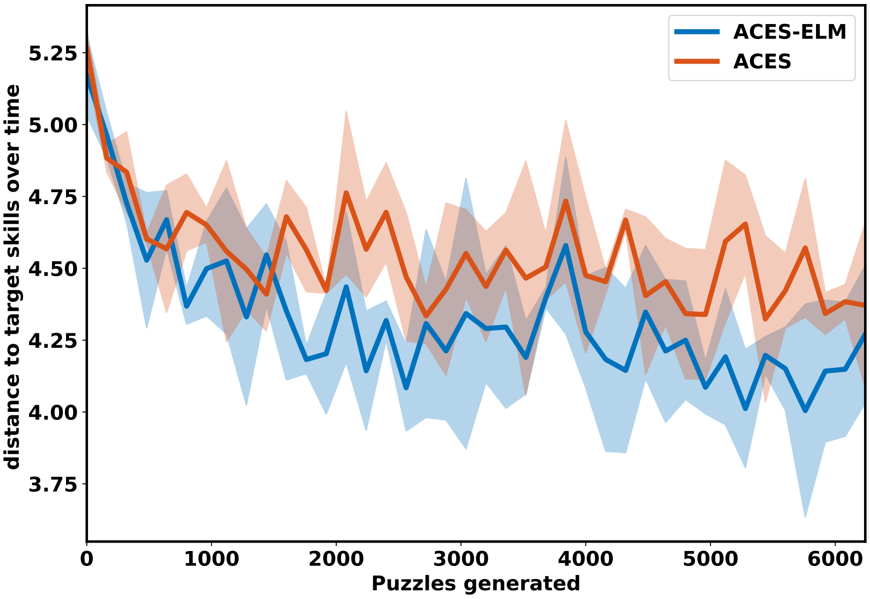


Figure 10: Distance to target skills over time.

Example generation for ACES

```python
from typing import List
def f(moves: List[List[int]], initial_state=[3, 3, 2, 2, 3,
    8]) -> bool:

    def bot_move() -> bool:
        vals = sorted(state, reverse=True)
        i_largest = state.index(vals[0])
        state[i_largest] -= max(vals[0] - vals[1], 1)
    state = initial_state[:]
    for (i, n) in moves:
        assert 0 < n <= state[i], 'Illegal␣move'
        state[i] -= n
        if set(state) == {0}:
            return True
        assert any(state), 'You␣lost!'
        bot_move()
def g(initial_state = [3, 3, 2, 2, 3, 8]):

    state = initial_state[:]
    moves = []

    def bot_move():  # bot takes objects from the largest heap
         to make it match the second largest heap
        vals = sorted(state, reverse=True)
        i_largest = state.index(vals[0])  # largest heap
        state[i_largest] -= max(vals[0] - vals[1], 1)  # must
            take some, take 1 in case of tie

    def losing(h):  # return True if h is a losing state
        xor = 0
        for i in h:
            xor ^= i
        return xor == 0

    def optimal_move():
        assert not losing(state)
        for i in range(len(state)):
            for n in range(1, state[i] + 1):
                state[i] -= n
                if losing(state):
                    moves.append([i, n])
                    return
                state[i] += n
        assert False, "Shouldn't␣reach␣hear"

    while True:
        optimal_move()
        if max(state) == 0:
            return moves
        bot_move()
```

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: While we do have broad motivations (open-ended problem generation), we make it clear in the abstract and intro that we restrict ourselves to programming puzzles with a certain format. We accurately report our main numerical findings.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations of the approach are discussed in a separate paragraph in the Discussion (Section 5).

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [NA]

Justification: No theory.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: We precisely detail the algorithms used, both our contribution and baselines, and provide all prompts used in the appendix with all model ids. While it should be possible to reproduce our data generation process from this information alone, the code is available at `https://github.com/Julien-pour/OpenELM/tree/imgep-qdaif`.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
    (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
    (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
    (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
    (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: The code is available at `https://github.com/Julien-pour/OpenELM/tree/imgep-qdaif`.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so “No” is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: All experimental details are shared in our Method and Result sections (seed dataset, temperature for generation, etc). The complete code will be released with the camera-ready version of the paper.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: Standard deviation is reported across seeds on our plots when appropriate.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: We documented the specs of the machines on which we performed our experiments at the beginning of our Results section. The computational costs of the development phase were not monitored, but we estimate they are negligible in comparison with the cost of our extensive experiments.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: We have reviewed the code of Ethics and do not find we have violated any principle in our research.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: We discuss broader impacts in our discussion section.

Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

    Answer: [NA]

    Justification: We do not release any pretrained LLM, image generator, or scraped dataset.

    Guidelines:

    - The answer NA means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: We prominently use the P3 dataset and the Llama 3 open model which are properly credited.

    Guidelines:

    - The answer NA means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.

- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [No]

    Justification: The code is released but not yet properly documented; we plan to add a cleaner version that is documented as soon as possible.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: No human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: No human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.