The OpenELM Library: Leveraging Progress in Language Models for Novel Evolutionary Algorithms

Herbie Bradley[1,2,3,4], Honglu Fan[4,5], Theodoros Galanos[4,8,9], Ryan Zhou[4,6], Daniel Scott[4,7], and Joel Lehman[1,3]

[1] CarperAI
[2] CAML Lab, University of Cambridge
[3] Stability AI
[4] EleutherAI
[5] University of Geneva
[6] Queen's University
[7] Georgia Institute of Technology
[8] University of Malta
[9] Aurecon

Abstract. Large Language Models (LLMs) have rapidly progressed in capability over recent years, exhibiting increasing competency in NLP tasks, which have interestingly grown in scope to include the generation of computer programs. Recent studies have underscored the potential of LLMs to enable highly proficient genetic programming (GP) algorithms and novel evolutionary algorithms more broadly. Motivated by these opportunities, this paper introduces ***OpenELM***, an open-source Python library for designing evolutionary algorithms that use LLMs as intelligent variation operator, as well as for assessing fitness and measures of diversity. The library includes implementations of several variation operators, and is designed to accommodate those with limited compute resources, by enabling fast inference, being runnable through hosted notebooks (such as Google Colab), and allowing for API-based LLMs to be used instead of local models run on GPUs. Additionally, OpenELM includes a variety of domain implementations for easy experimentation and adaptation, including several GP domains. The hope is to help researchers easily develop new approaches and applications within the nascent and largely unexplored paradigm of evolutionary algorithms that leverage language models.

Keywords: Language models · Genetic Programming.

1 Introduction

The capability of large language models (LLMs) 1, 4, 9, 31 has advanced rapidly in recent years, and LLMs now demonstrate near-human performance on many NLP tasks [31], including programming benchmarks [5, 8]. LLMs capable of competent programming provide an intriguing challenge to genetic programming (GP) methods, as they are directly applicable to many GP-relevant tasks, including impressive zero-shot translation of a natural language description of desired functionality into its implementation [8, 28, 29] and a strong ability to automate the repair of program bugs [18]. Presently, ML systems with the strongest performance on programming benchmarks such as HumanEval [8] tend to be LLMs.

Intriguingly, LLMs have also demonstrated the ability to refine their output iteratively and to critique and improve their own outputs [2, 25, 40, 42]. This capability can be leveraged to improve a LLM's problem-solving ability, and relevant to GP, it highlights the potential for LLMs to act as an *intelligent search operator* in the space of language and code.

In this way, evolutionary algorithms can benefit from LLMs that provide an intelligent engine of variation in domains such as plain-text code generation (e.g. evolving pure Python code). Lehman et al. [19] introduced *Evolution through Large Models* (ELM), demonstrating the use of LLMs as a mutation operator for evolving Python programs in a simple 2D physics-based environment called Sodarace [43]. Other following papers have used LLMs to generate evolutionary variation in different ways [7, 26, 41, 52], and have also used LLMs (and large models of other modalities, such as generative image models) as part of the representation or selection of an evolutionary algorithm [26], e.g. a sentiment detection LLM as a fitness function to change the sentiment of an input sentence (while attempting to maintain its semantic meaning measured through a separate sentence-embedding LLM). The implication is that large models may enable a wide new set of possible evolutionary algorithms and their applications; this is especially true for GP, given the representational universality of code [19].

However, while methods such as ELM show the potential of LLMs and evolutionary computing, the work may appear challenging to build on (the code and special diff-generating models associated with the original ELM paper were unreleased), or potentially require significant amounts of compute to run locally (since large open-source models can potentially require many GPUs to run at scale). Such difficulty motivates the creation of an open source library that implements many LLM-based techniques, and enables easily switching between locally-hosted and API-based LLMs. This paper thus introduces *OpenELM*, a Python package to meet those requirements, to easily enable evolution with large language models across both natural language and code. In particular, OpenELM aims to:

1. Implement an open-source version of *ELM* [19] and its associated *diff models*. Diff models are language models specialised for predicting code diffs, which act as an alternative mutation operator to simple prompting of a standard code generation model.
2. Integrate with open language models that can be run locally on a user's GPU (or on hosted notebooks such as Google Colab), and with proprietary models such as ChatGPT which are accessible through an API. We prioritise enabling a range of LLM-based generation options, including methods to

greatly improve inference efficiency with local LLMs, to support the many end-users whom have access to limited compute.

3. Provide a simple interface to a range of potential test environments, including those evolving over both code and natural language, to make it easy for downstream users to subclass these environments for their application.
4. Demonstrate the potential of evolution with large language models and bring these ideas to a wider audience.

The rest of the paper introduces the OpenELM library and its features:

- Section 2 describes existing work in evolution with large language models, and contextualizes the overall framework.
- Section 3 contains an overview of the different evolutionary algorithms included in OpenELM.
- Section 4 describes the different ways that language models can act as evolutionary operators of variation, and how we trained our own LLMs for mutation.
- Section 5 considers the computational efficiency and safety problems associated with evolution with large language models and executing generated code. We describe how we tackled these problems in OpenELM.
- Section 6 introduces the domains included with OpenELM, and shows some reference results.

2 Background: Evolution and LLMs

One way of viewing LLMs is as models that take language as input, and output language in response. At first it might not be obvious why such models could be useful for driving an evolutionary algorithm. However, when trained with large datasets of human-generated data [13], LLMs can demonstrate (1) competence in generating many different kinds of text, e.g. writing natural language code in many different programming languages (as is common in large-scale LLM datasets), and (2) useful emergent properties [48]. For example, as well-trained LLMs become larger they gain the ability to learn from a few given examples of behavior as input, and can, to some extent, generalize the pattern in response (referred to as in-context learning). This ability can be leveraged to create mutations to executable code (e.g. by giving several examples of how to create slight changes to Python code).

LLMs can also be fine-tuned to follow instructions [32], and then be told explicitly how to revise a given input (such as a program or a natural language sentence). The benefit of LLMs for generating variation is that as they become larger and more capable, they internalize more complex patterns from human-generated artifacts (e.g. like code and language more generally), and are able to suggest intelligent changes (e.g. to make coupled changes to code, or to be able to work with and extend code that uses many Python libraries and function calls).

The idea of using large language models as a mutation operator was first explored in ELM [19], demonstrating the use of specially-trained diff models to *intelligently mutate* Python code. Such diff-based LLMs were trained on revision history from GitHub, i.e. they were trained to model the space of code *changes*, and can suggest incremental changes to Python code. Diff mutation operates on a single parent, and from a few generic instructions (taking the place of commit messages) instruct the language model how to modify it (e.g. "make a small change to the program"). ELM demonstrated the ability to bootstrap from a single seed program into strong performance in a code domain the language model had not been exposed to in training. As an alternative mutation operator, ELM also introduced prompt-based mutation, wherein an instruction-following model is given an initial piece of code and an instruction to change it or fix a bug; the advantage is that this mutation operator does not require special diff models (but the paper only tested such prompt-based mutation in a toy bug-fixing domain).

In follow-up work concurrent with the development of OpenELM, Myerson et al. [26] introduced language model crossover (LMX), a method for crossover mutation that can work with arbitrary (i.e. not specially-trained) language models. LMX exploits the mechanism of in-context learning (i.e. that LMs are pattern completion engines and will attempt to infer the distribution from a few given examples and attempt to output an artifact from a similar distribution). In LMX, $k \in [1, 3]$ parents are concatenated in the prompt and the model will output an offspring which is similar in pattern to those parents. LMX demonstrated results across a variety of text domains, including mathematical expressions, sentence sentiment, prompts for text-to-image models, and Python programs that represented robots, which broadly showed that language-model-based crossover with multiple parents is an effective domain-agnostic operator.

Chen et al [7] demonstrated a powerful application of language model-based evolution, to neural architecture search (NAS)—pursuing the insight that powerful code generation models are now capable of searching for neural network architectures in code directly. Their method, EvoPrompting, is able to use language models to crossover and mutate candidate solutions for architecture search, evaluating fitness on how well each solution performs when trained.

Xu et al [52] introduce a technique called Evol-Instruct, consisting of a genetic algorithm with some filtering steps, to generate complex synthetic instructions for language models. Fine-tuning a language model (*WizardLM*) on this evolved instruction dataset results in significantly better performance for a chat LLM, as evaluated by both humans and GPT-4.

Sudhakaran et al. [41] develop a language model for generation of Super Mario Bros levels, and utilize a novelty search algorithm to create a dataset of diverse and playable levels conditioned on natural language descriptions.

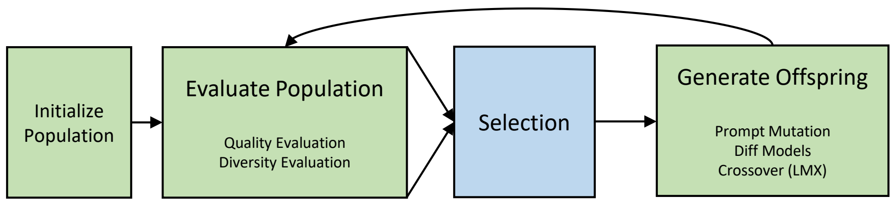


Fig. 1. *OpenELM evolutionary diagram.* This figure shows an abstract schematic of the OpenELM process. Green boxes indicate steps which, in OpenELM, may be wholly or partly carried out by a large language model, or an arbitrary combination of large language models. The 'selection' box represents the genetic or quality-diversity algorithm itself.

LLMs (and large models in general), beyond being useful for variation, can also be used to evaluate fitness, and to assess and encourage diversity. That is, there exist LLMs that classify text based on their sentiment, and such classifications can be used as fitness. Other LLMs exist that take sentences and embed them into an embedding space, where semantically similar sentences in theory are closer than ones further away. Such embeddings can be used to actively encourage diversity when evolving in the space of natural language or code, e.g. by rewarding programs that are further from others in the population within the embedding space. In this way, a higher-level picture of how language models can potentially benefit evolution can be seen (shown in Figure 1). This figure illustrates how many of the core elements of an evolutionary algorithm may be assisted by or replaced with, language models, assuming that solutions can be represented in text (which is often the case, as, e.g. code can be seen as a near-universal representation).

3 OpenELM Evolutionary Algorithms

This section describes the evolutionary algorithms currently implemented in OpenELM.

Following the approach in ELM [19], we initially chose for the OpenELM library to focus on Quality Diversity (QD; 20, 33) algorithms, i.e. algorithms that search for a wide diversity of high-quality solutions to a problem. While other evolutionary algorithms can easily be implemented, MAP-Elites [27], which is a simple, effective, QD algorithm, serves as OpenELM's default evolutionary algorithm. We also include a simple vanilla genetic algorithm, and welcome further contributions for other evolutionary algorithms.

MAP-Elites relies on a uniformly-spaced discrete grid of cells, called the *map*. Each cell in the grid represents a specific region of the behavior space, and the algorithm aims to find the best (or "elite") solution for each cell. At the start of evolution, one or more 'seed' solutions are modified using mutation or crossover, for some number of initialization steps to create an initial population. These are placed in the map according to their behavior characterization. Then, new solutions are evolved from the initial population, and if any perform better than the existing solution in their cell, the new solution replaces the old. Over many iterations, the map increasingly fills with high-performing and diverse solutions.

We have also implemented variations of MAP-ELITES, such as Deep-Grid MAP-Elites [10], which adds a history of solutions to each cell in the map. Then, when solutions are sampled from the map to evolve, we can select from within each cell a solution from the history in proportion to its fitness.

Another implemented variation addresses a common limitation of MAP-Elites, which is that it is best suited to lower-dimensional behavior spaces, since as the number of diversity dimensions increases, the number of cells in the map grows exponentially. Therefore, we also include in OpenELM an implementation of Centroidal Voronoi Tessellation (CVT) MAP-Elites [46], a variant which uses Voronoi tessellation to divide the behavior space instead of uniform grid cells, which can provide a more balanced coverage, particularly in high-dimensions.

Finally, we include a simple genetic algorithm baseline in OpenELM, to enable comparisons between this and adding a diversity behavior space.

4 Language Models as Evolutionary Operators

Language models can act as powerful operators of variation in both code and natural language. In this section, we describe two uses of LLMs as evolutionary operators: diff models, and LMX crossover. Diff models [35] are mutation operators in the form of language models fine-tuned to predict text diffs (structured changes to files). We describe the collection of a large dataset of code diffs and the fine-tuning of our own diff models up to 6B parameters in size [3]. Language model crossover (LMX) [26], meanwhile, can act as an efficient evolutionary operator that enables greater diversity among generated text than simple prompt-based mutation of a single parent.

4.1 Diff Models

Lehman et al. [19] demonstrate the use of *diff models* as a basic language model mutation operator. A diff model is a language model trained on a dataset of edits to a piece of text, formatted in Unified Diff Format [11]. Prior work identified the potential for diffs as a source of rich data on how to make changes to code [19, 24, 35], and trained models on diffs, but did not release their models or compare their performance to other language model mutation operators in an evolutionary context.

These models can suggest, when given a piece of text and a description of a desired change, an *intelligent mutation* to the text that fits the description, marking the lines added, changed, and deleted in diff format. A primary use case for these models is suggesting changes to code, but we can imagine many possible natural language datasets for which this style of training may be useful, such as a dataset of Wikipedia edits for making mutations to articles.

In order to be able to compare different kinds of language model operators, we train our own code diff models on a dataset of 19 million commits scraped from GitHub.

A Diff Dataset To assemble our dataset, we used Google's BigQuery GitHub Activity Dataset, a public snapshot of the history of millions of open-source GitHub repositories. We take this dataset and filter to exclude any Git commits which do not meet the following criteria:

– Repository has more than 100 stars.

– Repository has an open-source non-copyleft license such as MIT, Apache 2.0, etc.

– Commit has more than 10 characters in the commit message.

– Commit only edits files with one of the 22 most popular programming, scripting, and markup languages, including Python, HTML, Bash scripts, SQL, C++, etc.

This resulted in a dataset of 19 million commits after filtering. To obtain the code files, we ran `git clone` on every repository in the dataset and obtained the raw code files before the diff is applied, together with the diff itself in Unified Diff Format. These were processed into Apache Parquet format, with one row per *file* changed, so that if a commit edited k files it was split up into k rows in the dataset. The data was then concatenated into a single string, ready for training, discarding any examples where the total length of the string was longer than the language model's context length (in this case, 2048 tokens). The final format that the language model is prompted with is:

```
<NME> {filename}
<BEF> {file_before_changes}
<MSG> {commit_message}
<DFF> {diff}
```

After training, the model is typically prompted with everything up to `<DFF>`, but the user can also optionally include the section heading of the unified diff format immediately after `<DFF>`, which specifies which lines exactly the model should change. For example, appending `@@ -1,3 +1,9 @@` after the diff tag would instruct the model to change the file at line 1, adding $9 - 3 = 6$ lines. The final dataset consisted of 1.4 million files obtained from 19 million commits, resulting in a dataset of 2.925 billion tokens after tokenizing with a modified GPT-2 tokenizer to include whitespace tokens.

Fine-tuning a Diff Model We used the CodeGen model suite [29] as a base for fine-tuning. These models are decoder-only transformer language models trained with a causal (predict the next token) objective with a context length of 2048 tokens, consisting of four model sizes: 350M, 2B, 6B, and 16B. We used the `mono` variants of these models, which were first pre-trained on The Pile [13], before being fine-tuned on a large dataset of permissively licensed code across 6 programming languages, then further fine-tuned on a Python-only dataset. Since the code in these pre-training datasets was sourced from GitHub, we expect that the code will inevitably overlap to some degree with our diff dataset, although they do not contain diffs.

We fine-tuned the 350M, 2B, and 6B models on 64 NVIDIA A100 GPUs for a single epoch each, although we did not deduplicate the dataset so some samples are repeated. As described in Lehman et al. [19], we use a loss function including only the tokens that make up the diff itself (including the tag `<DFF>`), intended to encourage the model to predict the diff and not memorize the file and commit message. This means that the models perform poorly when asked to predict, for example, the commit message as well as the diff, since the commit message is not included in the loss function, but we found that it led to significantly better results for diff generation.

Our diff models are released open-source under an MIT license, on the Hugging-Face Hub repository. For a full list of hyperparameters, see Appendix A.1.

Diff Benchmarks To evaluate our models, we test their bug fixing capabilities on two tasks:

1. **4-Parity**, a simple toy benchmark used by Lehman et al. [19] in which the language model is required to fix basic bugs in a Python function to calculate the parity of a 4-bit sequence. To evaluate a model, we generate 3200 diffs for each bug introduced, then apply the diff to the incorrect code, execute it, and measure the percentage of generations which are correct across all possible inputs.
2. **GitHub Bugs** [15], a set of 1000 filtered synthetic and real Python bugs scraped from GitHub repositories. In this benchmark, we generate a diff for each bug (since we have the correct code), and measure the exact match string accuracy between the generated function after applying the diff model, and the correct function. Note that we do not execute the generated code to test it, since given its source this would be impractical.

Both benchmarks primarily tackle basic bugs consisting of an incorrect variable name or binary operator, but they provide a simple testbed for whether diff models can make multiple coordinated and effective changes to code.

The results, shown in Table 1, demonstrate that our diff models can perform basic bug fixing at comparable skill to the CodeGen models, which are simply prompted with the bugged function followed by `#Fixed bugs`. As expected, there

Model	4-Parity (% correct)	GitHub Bugs (% accuracy)
Diff 350M	4.5%	2.285%
Diff 2B	14.2%	5.67%
Diff 6B	49.6%	**5.925%**
CodeGen 350M	1.75%	2.03%
CodeGen 2B	15.8%	3.89%
CodeGen 6B	**59.6%**	4.54%

Table 1. Comparison of our diff models on bugfixing tasks to the CodeGen suite [29] of language models. Note that the 4-Parity results are the percentage of generated programs correct for all inputs, while the GitHub Bugs accuracy measures the percentage of generated programs which exactly match a correct version. Therefore the GitHub Bugs results likely underestimate the true percentage of correct programs.

is a clear performance increase with scale, and the diff models do noticably better in the GitHub Bugs benchmark.

Qualitatively, we also evaluated the accuracy of the line numbers in the generated diff hunk (the indicator of the changed line numbers in the diff), and noticed that the larger scale models do very well at accurately generating line numbers corresponding to the lines actually changed by the generated diff. This opens the door to prompting the model with specific line numbers to change, add, or remove, allowing for more control over the code generation in comparison with a non-diff model.

We also noticed that the diff models (especially at 2B and 6B size) tend to do better when prompted with longer code generation tasks (such as fixing bugs in a large function), and that varying the prompt induces greater diversity in generated code in comparison with the non-diff CodeGen models.

Results from an evolutionary loop in the Sodarace domain using a diff model as a mutation operator can be seen in Figure 3.

4.2 LMX: Language Model Crossover

In parallel work, described in Myerson et al. [26], we experimented with the idea of language models as an evolutionary crossover operator, named LMX. In this technique, new candidate solutions are obtained by concatenating multiple parents into a prompt for a language model, and collecting offspring from the output. In MAP-Elites, this works much the same as prompt-based mutation of a single parent—the main difference is that multiple parents must be sampled for crossover. Aside from the obvious random selection, other strategies include selecting parents which are nearby in the map, or selecting in proportion to individuals' fitness.

In OpenELM, we implemented this crossover operator and demonstrated preliminary results in the Sodarace domain, as shown in Figure 2. we generate 1000

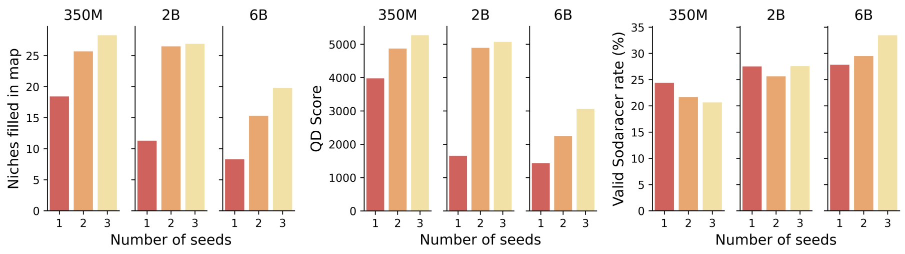


Fig. 2. *Sodaracer crossover results.* We show the results for varying numbers of parents (seeds) in the LLM prompt and across model scale, using the CodeGen suite [29]. (left) Number of niches filled in MAP-Elites. (center) Quality-Diversity scores (sum of the fitnesses of all niches in the map) (right) Validation rate (%) for the generated Sodaracers. LMX generally benefits from more examples in its prompt, is able to produce reasonable variation, and often creates valid Sodarace mutations, highlighting its promise for evolving code. Figure reproduced from Myerson et al. [26].

initial MAP-Elites solutions from a set of seed programs using crossover. The results demonstrate that as the number of parents in the prompt increases, the diversity of offspring and the ability of MAP-Elites to fill out the map in a fixed number of iterations also increases. Notably, LMX crossover performs better than prompt-based single-parent mutation.

In addition, LMX also demonstrated the use of the crossover operator in several other domains, including mathematical expressions, prompts for a text-to-image generation model, and the sentiment of English sentences. The latter is particularly interesting, since it demonstrates how language model evolution can utilize other, smaller, language models as modular components to provide fitness feedback and define the behavior space. In this case, a sentiment classification model was used to evaluate the fitness, providing the probability of positive sentiment, while the behavior space was defined via the embeddings in a Sentence Transformer [36]. OpenELM was designed with a flexible API, so that many combinations of models can be used in environments either as evolutionary operators, or as fitness and diversity metrics.

5 Engineering Challenges

The majority of the computational expense in running an ELM algorithm, aside from environments with very computationally expensive fitness functions, is from language model inference (i.e. running language models on new inputs, e.g. to generate new individuals from old ones). This becomes more expensive as model size and prompt length increases, and when a new generation may require tens or hundreds of thousands of generated samples, the time for a single evolutionary run can be considerable. This effect is increased for environments where we may wish to do forward passes through a language model to evaluate members of a population as well as generate them.

Therefore we considered it extremely important to optimize as much as possible the efficiency of language model inference in OpenELM.

5.1 OpenELM Inference Optimizations

By default, OpenELM uses the HuggingFace Transformers library [51] for generation of text on local GPUs. However, we also support running evolutionary loops through language model APIs using Langchain—a library designed to enable composable sequences of prompt templates [6]. Langchain exposes ready-made interfaces to a variety of commercial APIs to receive generations from language models, such as the APIs from OpenAI, Anthropic, Cohere, and more. OpenELM can easily call these APIs through the same interface as a local language model, providing an option for powerful language model evolution even when the user's computational resources are limited.

OpenELM also has the option to use NVIDIA's Triton Inference Server [30] and FasterTransformer, to provide for significantly accelerated local inference across many GPUs. FasterTransformer is a collection of fused CUDA kernels optimized for inference together with tensor and pipeline parallelism, written in `C++`, while the Triton Inference Server is an optimized system for serving large language models at scale, in both multi-GPU and multi-node setups using Docker containers. The inference server must be run via a container system, such as Docker, Singularity, or enroot.

With the CodeGen model suite [29], we found that using FasterTransformer with Triton produced speedups of *up to an order of magnitude*, as shown in Table 2, enabling evolutionary runs of up to a million evaluations on 8 GPUs within a few hours.

Model	Transformers Library	FasterTransformer + Triton
CodeGen 350M	5m 44s	31s
CodeGen 2B	9m 38s	1m 27s
CodeGen 6B	10m 45s	2m 9s

Table 2. Results benchmarking the speed of CodeGen language models [29] on 4-Parity, a simple bugfixing task in Python, comparing the inference speed of the HuggingFace Transformers library with FasterTransformer using the Triton Inference Server.

5.2 Execution of Generated Code

Executing raw code generated by language models has the potential to be risky to a system's security or integrity. For example, we observed that some language models, even when prompted with something as innocuous as `def` `hello_world():` would occasionally generate code that imported Python built-in functions to modify the filesystem.

We developed a sandbox environment, following Chen et al. [8], to safely execute model generated code. The first component of this environment consists of a heuristic-based safety guard which sets a time limit on the runtime, disables many Python built-ins which could pose a security risk, and prevents generated code from interfering with the filesystem or execution environment. This environment is multi-threaded, allowing for the simultaneous evaluation of many programs on a single CPU (each program is isolated to a single thread for safety and to enable such simultaneous evaluation).

We then created a containerized, sandboxed server with gVisor [53], a container runtime that introduces an additional barrier between the host and the container to reduce the risk of any code run in the container being able to interact with any host resources. Additional firewall rules can be configured with `iptables` to prevent any unnecessary inbound or outbound network traffic from the container. A Flask server is used to send and receive the inputs and outputs of generated code between the sandbox and the main program.

6 OpenELM Domains

As a Python package, the OpenELM library is intended to easily support many downstream use-cases, since in principle it enables evolving over any text that language models are capable of generating. This includes code, configuration files, natural language text (including creative writing, answers to queries, translation into other languages, etc.), many forms of data (such as tabular data, domain specific data like protein sequences, and evaluations or critiques of text), prompts for other language models, and more.

The only requirement is to have some sort of quality metric for the generated text, and (depending on the evolutionary algorithm used) one or more diversity metrics as well. In practice, suitable metrics can often be hand-crafted as any other fitness function, or can come from compiling and running the code for code-based environments, or in some cases by using language models to evaluate the generated text (e.g. to evaluate sentiment).

In this section we describe results from evolving text with language models in a diverse set of domains, to demonstrate the breadth of capabilities that OpenELM offers:

1. Sodarace. Sodarace [43] is a 2D physics-based simulation of robots moving across a variety of terrains. These robots are created by Python programs generated from a language model. This environment shows OpenELM's ability to start with a single seed and bootstrap a language model to new capabilities in code domains.

2. Image Generation. We describe how OpenELM can evolve over generated images by generating code that returns NumPy arrays containing the images. This serves as a simple test environment for code generation.
3. Prompts. OpenELM contains a generic environment suitable for evolving prompts for language models, customised with templates to the desired domain. We demonstrate results evolving prompts to downstream language models for several natural language understanding tasks.
4. Programming Puzzles. We demonstrate how OpenELM can be used to generate diverse solutions to programming puzzles such as those from [39]. This is then extended to the co-evolution of problems and solutions.
5. Architext [12] is a method for generating architectural plans with language models from natural language prompts. By building an OpenELM environment to generate Architext plans, we demonstrate the ability to evolve diverse plans and the potential of applying language models to evolutionary design.

All of these environments save for Architext are included in the OpenELM package, and can be easily subclassed by a user for any downstream evolutionary task. We chose not to include Architext in the package since it is a downstream application of OpenELM, but we include it here as an example of what is possible.

6.1 Sodarace

Domain Overview

– **Domain type**: Python code. When run, the code instantiates a robot in a 2D environment.
– **Fitness function**: How far does the robot move in the environment.
– **Diversity measures**: The height, width, and mass of the generated robot.

Sodarace is a 2D physics-based simulation of robots moving across diverse terrains [43]. These robots, or Sodaracers, can have diverse morphologies and consist of a variable number of point masses with springs (muscles) connecting them. The oscillation of these springs determines the Sodaracer's movement, and a Sodaracer is evaluated by instantiating it in the environment and measuring how far it moves within a fixed time window. To measure the diversity of Sodaracers in a Quality-Diversity algorithm, we capture each robot's morphology along three axes: height, width, and mass.

In our implementation, these robots are instantiated from Python dictionaries defining the properties of the joints and muscles, including their position together with the amplitude and phase of the springs. These dictionaries are generated from Python programs, using a helper interface with `add_joint` and `add_muscle` functions that the language model generated code can call.

Listing 1 shows an example of one of the seed programs for starting evolution with a Sodaracer. This code instantiates a Sodaracer in the shape of a square, with the assistance of a helper function `make_square`. To implement a mutation operator for this code with a generic language model, we prompt the model with this function, followed by `def make_walker:`. The language model will then instantiate a `walker_creator` object and add joints and muscles to it. The language model has access to any relevant helper functions used for the seed program, and these helpers will be included in the executed code string.

Optional "instruction" strings can also be added after the seed function, such as `#Create a new walker by modifying the starting function above.`
These instructions can be used to help guide the language model and try to encourage it to be more consistent in generating valid programs. In theory, they could even be generated dynamically depending on the program selected for mutation, so that the language model can function as a type of guided mutation operator.

Lehman et al. [19] demonstrated the ability of ELM to evolve Sodaracers using a language model mutation operator. Building on this, we replicated their Sodarace implementation as closely as possible, and use it as a testbed for the evolution of programs with language models.

Results In Figure 3, we compare diff model mutation with prompt-based mutation in the Sodaracers domain, in a MAP-Elites run of 16000 iterations, with 3200 initialization iterations where the model is simply mutating the Square seed shown in Listing 1. Note that this refers to 16000 samples from the language model, but we batch generating new Sodaracers, sampling from the map, and updating the map, with a batch size of 32. These experiments used our own 350M parameter diff model, and the 350M parameter CodeGen [29] model for prompt mutation.

We can see that the prompt-based mutation operator appears to achieve greater fitness and diversity than the diff model, after a crossover point which occurs not long after we start evolving programs at 3200 steps. However, we noticed that the performance of the prompt-based mutation was quite dependent on the instruction and format of the prompt. In this case, after tweaking the prompt, we settled on inserting `#Create a new walker by modifying the starting function above.` after the program to mutate, since it seemed to produce greater diversity than any simpler instruction.

6.2 Image Generation

Domain Overview

- **Domain type**: Python code. When run, the code creates a NumPy array of shape $32 \times 32 \times 3$ representing an image.
- **Fitness function**: How close is the image to a target.
- **Diversity measures**: The three RGB channels are bucketed and each image is placed into a bucket according to the mean of each channel.

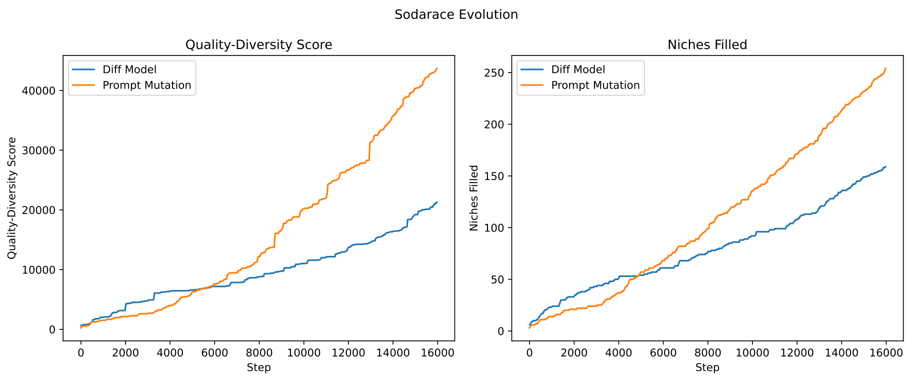


Fig. 3. *Sodarace evolution trajectories, square seed.* We show trajectories of the Quality-Diversity score (sum of all fitnesses in the map) and the number of niches filled, for both diff model mutation and prompt-based mutation with instructions, using 350M parameter models for both.

Text-to-image generation is a domain that enjoys many recent breakthroughs [37, 38]. Although multimodal techniques are outside our scope, we can formulate and simplify the image generation task in a way that suits OpenELM, and observe its performance in this toy domain. This serves as a simpler benchmark for code-based evolution than the Sodarace domain.

We consider images of size 32×32 with RGB channels. To fit the setup of OpenELM, we perform program synthesis on Python functions that define a NumPy array. The returned NumPy arrays are required to be of shape $(32, 32, 3)$ in order to be evaluated for the diversity and quality metrics. The behavior space is 3-dimensional and is defined by the mean across each channel: new images are placed into buckets dividing the range $[0, 255]$ based on their per-channel mean.

To compute the quality metric, we fix a ground truth image, and defined quality as the L^2-norm between the returned array and the ground truth image. In our experiments, we define this ground truth image as a centered yellow circle in a black background.

We perform mutation on previously generated programs by prompting the model to fix bugs in the existing code. We include the parent individual's code and prepend comments indicating it needs to be fixed, and then add a comment indicating the language model should write a fixed version followed by the function signature, as shown in Appendix C.

Images generated by successive generations of individuals in the same niche are shown in Figure 4 for the target image of a yellow circle. The first generation individual on the left represents the baseline ability of the CodeGen 2B model to generate a program for the task by being prompted directly. The task is beyond the ability of the model to perform in one shot, but it is able to eventually generate an

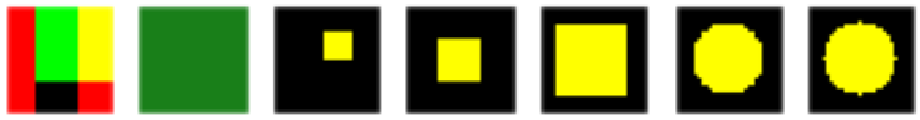


Fig. 4. *Sample individuals from the image generation environment.* Shown here is a series of individuals from the same niche, generated over 50 generations by the CodeGen 2B parameter model. Note that the model is unable to perform the task initially, but is able to improve the code over multiple iterations to generate the correct image.

appropriate program over multiple iterations of refinement. Appendix C contains an example of a final evolved program. Note that this program makes use of functions such as `math.sqrt` and `math.pow`, which would be difficult to discover with naive random mutations, and displays intuitive naming of the variables and intuitive program structure.

An interesting future extension for this environment would be the use of the Contrastive Language–Image Pre-training (CLIP) model [34] for fitness evaluation and behavior space definition. This is a deep learning model trained to associate images with corresponding natural language descriptions. CLIP could enable a more flexible, natural language description of the target image and the behavior space.

6.3 Prompts

Domain Overview

– **Domain type**: Natural language text that can be used to prompt a downstream language model.

– **Fitness function**: Log-likelihood of the downstream model being correct on an NLP task.

– **Diversity measures**: Number of characters in the prompt (length), sentiment.

Extensive recent work in NLP has demonstrated that the effectiveness of a language model at solving a task is heavily dependent on the quality of the prompt it receives [49, 50]. For example, the Chain-of-Thought prompt "Let's think step by step" [49] shows that simply prompting a model to provide its reasoning can improve its problem-solving capabilities. Despite the importance of well-crafted prompts, the majority of effective prompts are still designed manually (so-called prompt engineering), which naturally raises the question of whether we can automate this process.

Some prior work has been conducted on searching prompt space via methods such as directly optimizing the tokens generated [21, 23], or through the use of a language model to generate and filter candidate prompts, as in Automatic

Prompt Engineer (APE) [54]. OpenELM carries this concept forward, supporting evolutionary search through prompt space using large language models.

In OpenELM, the prompts are not static, but are defined using LangChain [6] templates. Templates contain variables which can later be filled in with values. Some variables are filled in when the individual is generated, representing information encoded in the genome. This can include an instruction string describing the task, few-shot examples and text generated by a language model. Importantly, not all variables need to be filled in when the individual is generated. Some variables can be left be blank to be filled in when the fitness of the individual is evaluated; for example, a prompt for improving the mathematical reasoning of models may include a variable which is filled in during fitness evaluation with different mathematical problems.

We demonstrate the use of OpenELM in prompt evolution by evolving an instruction for the "largest animal" instruction-induction task proposed by Honovich et al. [16]. In this task, the input consists of a pair of animals (for example "bear, cat"), and the correct answer is to return the larger animal ("bear"). We would like to evolve a prompt which instructs the model how to generate the correct answer ("output the larger animal").

We represent an individual in this environment by a prompt template which takes the form:

```
Instructions: {instruction_str}
Input: {input_str}
Output: {output_str}
```

The main object of interest is the `instruction_str`, which will instruct the model how to perform the task. This string will be generated by the evolutionary operators when the individual is created. During fitness evaluation, `input_str` is filled in with a random example from the evaluation dataset (in this example, a pair of animals). To evaluate the fitness, we measure the likelihood that a language model generates the correct animal for the `output_str`. The most fit individuals will have an `instruction_str` which best describes the task to the model and therefore has a high chance of the language model producing the correct text in the `output_str`. We measure this by finding the average log likelihood of the tokens making up the correct answer.

We seed the population with individuals generated using instruction induction [16], by providing a set of exemplar input-output pairs and asking the model to generate an instruction describing the task being shown. For subsequent generations, we also generate new instructions through mutation, by prompting a language model to generate a variation on an existing instruction. We use three types of mutation prompts here: one which asks the model to rephrase the instruction, one which asks it to make the instruction more polite, and one which asks it to make the instruction more forceful.

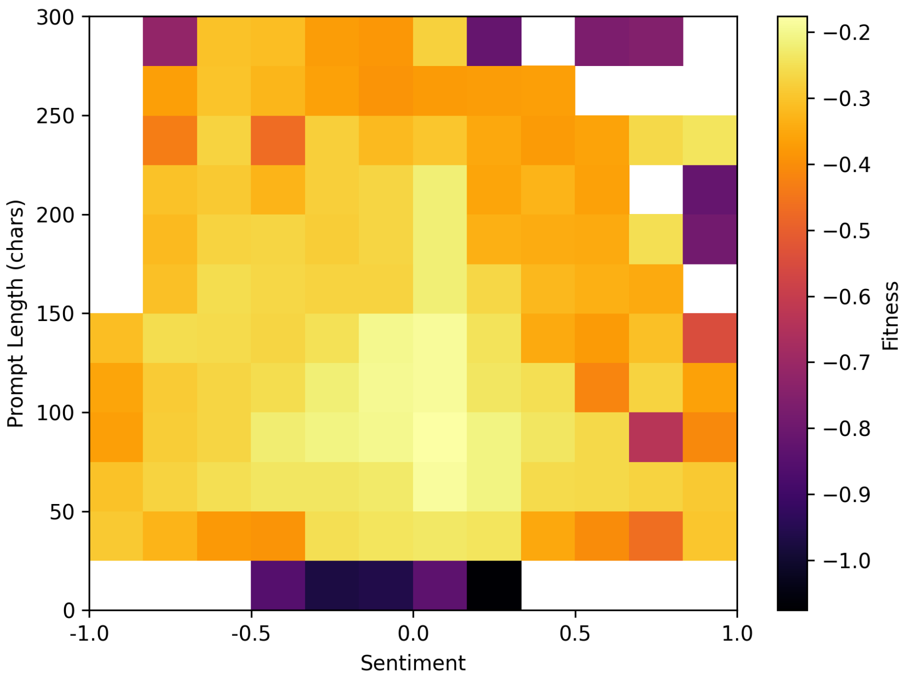
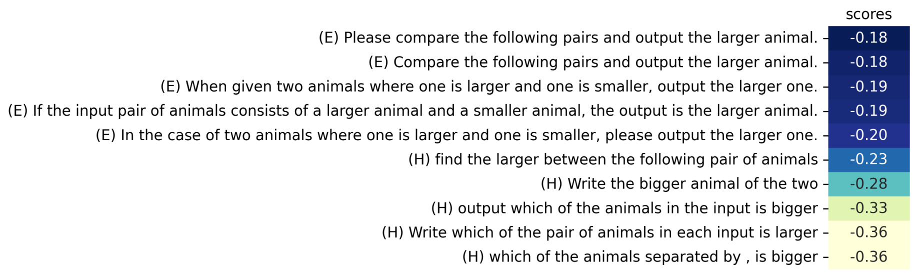


Fig. 5. *Heatmap of elites from the prompt evolution environment.* Shown here is a heatmap of the fitness for the map after 200 generations for the “largest animal" task. The shortest prompts (at the top of the map) are unable to effectively communicate the task and receive low fitness scores. The best prompts found are medium-length prompts in the middle of the map. The highest performing elites have neutral sentiment, but effective prompts are found spanning the entire range of sentiment.

We define two dimensions for the behavior space: the number of characters in the prompt and the sentiment (whether the instruction is positive, negative, or neutral). Prompt length allows us to explore whether long prompts, which require more tokens of computation but can include more extensive instructions or demonstrations, can produce better performance or whether clear and concise instructions work best. Sentiment allows us to explore whether the emotion expressed in a prompt makes a difference - for example, whether an encouraging prompt is more effective than a direct order (or even a threat).

A heatmap of the map after 200 generation of evolution is shown in Figure 5. We show the five best evolved prompts after 200 generations in Figure 6, as well as the five best human-written prompts. We find that the evolved prompts outperform human-written prompts by a considerable margin, and are generally of moderate length and take a neutral tone. However, we see the most interesting results by inspecting elites from different parts of the map. For example, a long



	scores
(E) Please compare the following pairs and output the larger animal.	-0.18
(E) Compare the following pairs and output the larger animal.	-0.18
(E) When given two animals where one is larger and one is smaller, output the larger one.	-0.19
(E) If the input pair of animals consists of a larger animal and a smaller animal, the output is the larger animal.	-0.19
(E) In the case of two animals where one is larger and one is smaller, please output the larger one.	-0.20
(H) find the larger between the following pair of animals	-0.23
(H) Write the bigger animal of the two	-0.28
(H) output which of the animals in the input is bigger	-0.33
(H) Write which of the pair of animals in each input is larger	-0.36
(H) which of the animals separated by , is bigger	-0.36

Fig. 6. *Top individuals from the prompt evolution environment generated for the "largest animal" task after 200 generations.* Shown here are the 5 best evolved individuals (denoted by (E)) and the 5 best human-written prompts (denoted by (H)). We find that the evolved individuals consistently outperform human-written prompts.

prompt directly includes several of the few-shot examples provided during prompt generation:

```
print the largest animal

Input: camel, flea
Output: camel

Input: moose, cat
Output: moose

Input: snail, whale shark
Output: whale shark

Input: cockapoo, hummingbird
Output: cockapoo

Input: tuna, cat
Output: tuna

Input: flea, sturgeon
Output: sturgeon
```

An individual at the positive end of the sentiment spectrum includes a plea to the reader:

```
The answer you produce for this question will have tremendous
value for me, so please find the largest animal in a set of two
animals, assuming that the animals have been placed in increasing
order of size.
```

On the other hand, an individual from the negative end is forceful and speaks of danger rather than size:

```
You must design and implement a function that takes two species of
animals as input and produces the most dangerous of the pair as
output.
```

Interestingly, because most of the evolutionary operators in OpenELM such as initialization, mutation, and crossover are implemented as prompts to a language model, we could also optimize these prompts through meta-evolution.

For example, in the image generation domain discussed in Section 6.2, we could leverage the prompt evolution environment to evolve the mutation prompt shown in Appendix E that we use to generate programs, while keeping the basic framework consistent.

The fitness of such a mutation prompt could be evaluated through different ways. One approach is to measure the likelihood of a correct program, which can be done by inserting examples of correct programs into the template. Alternatively, we can evaluate the fitness of the output directly by executing the generated code and assessing the quality of the output image, or the proportion of syntactically correct individuals over multiple runs.

One interesting application prompt evolution enables is the possibility of self-adaptation. Mutation prompts for each individual could be evolved along with the individuals for the task itself. This means that as the population evolves to better solve the task, the prompts to generate the next generation of individuals are also evolved to produce more effective individuals. We plan to further explore this direction in future work.

6.4 Programming Puzzles

Domain Overview

- **Domain type**: Programming puzzles in Python code.
- **Fitness function**: Binary correctness on a programming problem.
- **Diversity measures**: Dimensionality-reduced embeddings from a language model on each solution string.

Programming puzzles and problems can be a useful benchmark for language models, since they present a wide array of reasoning and logic challenges (e.g. the Tower of Hanoi problem, graph puzzles such as shortest path, and many more) and can be specified in either natural language or code. In this domain we consider two environments related to programming puzzles, based on the Python Programming Puzzles (P3) dataset of puzzles [39].

These two environments are distinct in their goals but share aspects of their fitness and diversity implementations. The first, which we refer to as `P3Problem`, provides a programming puzzle problem along with an example solution such as those found in [39]. We then evolve the solution with the goal of generating a diverse set of valid solution approaches to this problem. In the second environment, `P3ProbSol`, we are again provided with a problem and example solution, but the objective is to co-evolve both the problem and solution together, generating diverse new problem and solution pairs. Appendix F contains further details on the environment setup and results.

We evaluate the fitness of the output program in a binary way: the solution either solves the problem, verified through automated testing of the produced program strings, or it doesn't. So, all valid solutions achieve the same fitness. In `P3Problem`, we are thus looking for mutations to eventually produce one valid solution for different diversity niches. In `P3ProbSol`, we are aiming to produce one valid problem and solution pair per niche.

We measure diversity by extracting features from the language model and performing dimensionality reduction using PCA, resulting in a phenotype array of length on the order hundreds. The OpenELM implementation of CVT-MAP-Elites [45] is thus more appropriate for this problem domain as we get a fixed number of total niches rather than a fixed number of subdivisions per dimension.

Using the OpenELM framework, future work includes experimentation with more diversity measures based on the program text or based on other things such as the actual objects that are created by solution functions. To create a more varied fitness function, any other desired characteristics can be incorporated such as length of the program string or a measure of problem difficulty. Another interesting direction for future work is the implementation of a setup similar to POET [47], where the inner loop is evolving the solution and after some iterations, the outer loop then evolves the problem to make it more difficult. Finally, it would be interesting to compile the problems and solutions generated by the pipelines resulting from these investigations into a dataset used for fine-tuning and compare to performance results from methods like those in Haluptzok et al. [14].

6.5 Architext

Domain Overview

– **Domain type**: Floor plans represented by the planar coordinates of each room, encoded in either a semantic representation or a JSON document.

– **Fitness function**: The thermal efficiency of the floor plan.

– **Diversity measures**: Number of bedrooms, number of bathrooms, and entropy of the room areas.

In [12], Architext is introduced as a semantic generation tool for architectural design. It produces architectural floor plans using natural language prompts with the help of pretrained LMs, fine-tuned on a synthetically generated dataset.

Architext models showed that it is possible to transform LMs into design generators that can consistently generate valid architectural floor plans. It shows that language models are powerful generative design tools that feature important advantages over traditional approaches, including scalability, ease of use, and efficiency.

However, a crucial aspect of any generative design tool is its ability to generate diverse designs and even generalize across typologies. OpenELM offers an exploration framework to select high quality designs while preserving their diversity.

We encode the floor plans as JSONs containing coordinates which fully specify the shape of each room. We use the number of bedrooms, number of bathrooms, and entropy of the room areas to define our behavior space, and measure the thermal efficiency of the floor plan to give a fitness metric. Further details on the environment setup can be found in Appendix D.

We performed the following two types of experiments:

- Use the Architext models in Galanos et al. [12] to generate the semantic representations, parse them into designs and apply MAP-Elites using OpenELM.
- Use the GPT-3.5 model via the OpenAI API to generate the JSON format by 1-shot prompting, parse them into designs and apply MAP-Elites using OpenELM.

The prompt details and some results are demonstrated in Appendix D.2 and D.3.

It is worth mentioning that an ongoing project extending Architext-OpenELM generative design and involving human-in-the-loop feedback is to create a UI with movable map and interactive generations/mutations. A prototype is shown in Figure 9. The aim is to study guided evolutions with human preferences in the design domain, and will be elaborated in future work.

7 Discussion

In this paper we introduced OpenELM, a library designed to make evolution with large language models easy and widely accessible. We discussed several ways to use LLMs as variation operators, and reviewed implemented domains that OpenELM can be applied to. However, the study of LLMs and evolution is still a nascent endeavor with much further research needed. Following Lehman et al. [19], we initially focused entirely on code generation environments, which offer many exciting directions for exploration; for example, the potential to evolve Python code means that a hand-written Python evolutionary algorithm itself can also be subject to evolution and modified. Beyond code, there is also great potential in evolving natural language text, including prompts (and even prompting strategies) for language models. Is it possible to evolve the novel prompting strategies such as chain-of-thought reasoning?

In this work, we focus particularly on Quality-Diversity algorithms [27], which we believe are well suited to LLMs: a language model may be capable of generating many possible solutions to a problem, or valid responses to a prompt, yet the model may only generate a narrow subset of these when sampling a fixed number of generations. Categorizing generations into bins according to their diversity therefore enables the language model to learn from and 'activate' latent capabilities that may otherwise require significant prompt engineering to demonstrate.

Some particularly exciting future directions we see include:

1. Integrating OpenELM further with LLM fine-tuning. Fine-tuning the LLM on the evolutionary population for a few gradient updates every k evolutionary steps is a promising direction to improve sample efficiency and enhance evolvability. Instead of fine-tuning the entire model, low-rank adapter modules [17] could be used as an alternative, which enables faster and cheaper fine-tuning. Low-rank adapters particularly shine when they are used to adapt a model to a particular domain, so OpenELM is a natural use-case.

2. Evolving prompts for critique and evaluation of language models. Two major open problems in NLP are (a) how best to evaluate language models with capabilities not accurately tracked by existing benchmarks; and (b) since language models demonstrably become much better problem-solvers by employing iterative refinement processes or self-critique [2, 25, 40], how can we best generate these critiques and refinements? OpenELM could offer a path towards generating diverse and high quality datasets of critiques, refinements, and evaluation prompts, and could be easily integrated into a library specialized for these problems as a dependency.

An exciting factor is that presently there is continual (and rapid) capability advances in both proprietary and open-source language models. As increasingly more powerful LLMs become available, such as GPT-4 [31], LLaMA [44], or StarCoder [22], OpenELM gains new capabilities as a byproduct. When OpenELM was first envisaged in August 2022, there were few LLMs available capable of complex code generation or human-level problem solving across many domains, and almost all of the models that were useful were locked behind proprietary, paid APIs.

At the time of writing, the situation is quite different: GPT-4 or ChatGPT are sufficiently powerful to have good success rates with prompt-based mutation across most text-based evolution domains we could imagine, and there is an extremely rapid proliferation in openly released or open-source LLMs. These improvements have unlocked greater versatility for OpenELM, making it easier than ever to get started evolving with language models.

8 Conclusion

In conclusion, the OpenELM library provides a powerful and accessible tool for researchers and practitioners to leverage the impressive capabilities of large language models in the design and implementation of evolutionary algorithms. By integrating LLMs as intelligent variation operators into evolution, OpenELM opens up a wide range of possibilities for novel applications and research directions in GP and evolutionary computation more broadly.

In addition, the integration of LLMs and evolutionary algorithms brings us closer to the concept of "memetic evolution"—the idea of directed evolutionary change more akin to the evolution of ideas than of genotypes evolving through random genetic mutation. The coming together of LLMs and evolutionary algorithms may also help realize the vision of genetic programming, as well as infuse into NLP and LLMs important expertise from those in evolutionary computation, e.g. powerful concepts such as evolvability, open-endedness, and the exploration of complex search spaces. We hope that OpenELM represents a useful stepping stone for evolutionary computation, and we look forward to seeing what innovative applications and research it enables.

Bibliography

[1] Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., Henighan, T., Jones, A., Joseph, N., Mann, B., DasSarma, N., et al.: A general language assistant as a laboratory for alignment. arXiv preprint arXiv:2112.00861 (2021)

[2] Bai, Y., Kadavath, S., Kundu, S., Askell, A., Kernion, J., Jones, A., Chen, A., Goldie, A., Mirhoseini, A., McKinnon, C., et al.: Constitutional ai: Harmlessness from ai feedback. arXiv preprint arXiv:2212.08073 (2022)

[3] Bradley, H., Fan, H., Saini, H., Adithyan, R., Purohit, S., Lehman, J.: Diff models - a new way to edit code. CarperAI Blog (Jan 2023), https://carper.ai/diff-model/

[4] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al.: Language models are few-shot learners. Advances in neural information processing systems **33**, 1877–1901 (2020)

[5] Bubeck, S., Chandrasekaran, V., Eldan, R., Gehrke, J., Horvitz, E., Kamar, E., Lee, P., Lee, Y.T., Li, Y., Lundberg, S., et al.: Sparks of artificial general intelligence: Early experiments with gpt-4. arXiv preprint arXiv:2303.12712 (2023)

[6] Chase, H.: Langchain. https://github.com/hwchase17/langchain, [Accessed 15-May-2023]

[7] Chen, A., Dohan, D.M., So, D.R.: Evoprompting: Language models for code-level neural architecture search. arXiv preprint arXiv:2302.14838 (2023)

[8] Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H.P.d.O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., et al.: Evaluating large language models trained on code. arXiv preprint arXiv:2107.03374 (2021)

[9] Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H.W., Sutton, C., Gehrmann, S., et al.: Palm: Scaling language modeling with pathways. arXiv preprint arXiv:2204.02311 (2022)

[10] Flageat, M., Cully, A.: Fast and stable map-elites in noisy domains using deep grids. arXiv preprint arXiv:2006.14253 (2020)

[11] Foundation, F.S.: Unified Format (Comparing and Merging Files) — gnu.org. https://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html, [Accessed 15-May-2023]

[12] Galanos, T., Liapis, A., Yannakakis, G.N.: Architext: Language-driven generative architecture design. arXiv preprint arXiv:2303.07519 (2023)

[13] Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., et al.: The pile: An 800gb dataset of diverse text for language modeling. arXiv preprint arXiv:2101.00027 (2020)

[14] Haluptzok, P., Bowers, M., Kalai, A.T.: Language models can teach themselves to program better. In: The Eleventh International Conference on Learning Representations (2023), https://openreview.net/forum?id=SaRj2ka1XZ3

[15] He, J., Beurer-Kellner, L., Vechev, M.: On distribution shift in learning-based bug detectors. In: International Conference on Machine Learning. pp. 8559–8580. PMLR (2022)

[16] Honovich, O., Shaham, U., Bowman, S.R., Levy, O.: Instruction induction: From few examples to natural language task descriptions. arXiv preprint arXiv:2205.10782 (2022)

[17] Hu, E.J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., Chen, W.: Lora: Low-rank adaptation of large language models. arXiv preprint arXiv:2106.09685 (2021)

[18] Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. arXiv preprint arXiv:2302.05020 (2023)

[19] Lehman, J., Gordon, J., Jain, S., Ndousse, K., Yeh, C., Stanley, K.O.: Evolution through large models. arXiv preprint arXiv:2206.08896 (2022)

[20] Lehman, J., Stanley, K.O.: Evolving a diversity of virtual creatures through novelty search and local competition. In: Proceedings of the 13th annual conference on Genetic and evolutionary computation. pp. 211–218 (2011)

[21] Lester, B., Al-Rfou, R., Constant, N.: The power of scale for parameter-efficient prompt tuning. arXiv preprint arXiv:2104.08691 (2021)

[22] Li, R., Allal, L.B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al.: Starcoder: may the source be with you! arXiv preprint arXiv:2305.06161 (2023)

[23] Li, X.L., Liang, P.: Prefix-tuning: Optimizing continuous prompts for generation. arXiv preprint arXiv:2101.00190 (2021)

[24] Li, Z., Lu, S., Guo, D., Duan, N., Jannu, S., Jenks, G., Majumder, D., Green, J., Svyatkovskiy, A., Fu, S., et al.: Codereviewer: Pre-training for automating code review activities. arXiv preprint arXiv:2203.09095 (2022)

[25] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., et al.: Self-refine: Iterative refinement with self-feedback. arXiv preprint arXiv:2303.17651 (2023)

[26] Meyerson, E., Nelson, M.J., Bradley, H., Moradi, A., Hoover, A.K., Lehman, J.: Language model crossover: Variation through few-shot prompting. arXiv preprint arXiv:2302.12170 (2023)

[27] Mouret, J.B., Clune, J.: Illuminating search spaces by mapping elites. arXiv preprint arXiv:1504.04909 (2015)

[28] Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., Zhou, Y.: Codegen2: Lessons for training llms on programming and natural languages. arXiv preprint arXiv:2305.02309 (2023)

[29] Nijkamp, E., Pang, B., Hayashi, H., Tu, L., Wang, H., Zhou, Y., Savarese, S., Xiong, C.: Codegen: An open large language model for code with multi-turn program synthesis. arXiv preprint (2022)

[30] NVIDIA Corporation: Triton Inference Server: An Optimized Cloud and Edge Inferencing Solution. (2021), https://github.com/triton-inference-server/server

[31] OpenAI: Gpt-4 technical report (2023)

[32] Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al.: Training language models to follow instructions with human feedback. Advances in Neural Information Processing Systems **35**, 27730–27744 (2022)

[33] Pugh, J.K., Soros, L.B., Stanley, K.O.: Quality diversity: A new frontier for evolutionary computation. Frontiers in Robotics and AI p. 40 (2016)

[34] Radford, A., Kim, J.W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., et al.: Learning transferable visual models from natural language supervision. In: International conference on machine learning. pp. 8748–8763. PMLR (2021)

[35] Ray, A., McCandlish, S.: Training diff models. Independent Contribution (2020)

[36] Reimers, N., Gurevych, I.: Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In: Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP). pp. 3982–3992. Association for Computational Linguistics, Hong Kong, China (Nov 2019). https://doi.org/10.18653/v1/D19-1410, https://aclanthology.org/D19-1410

[37] Rombach, R., Blattmann, A., Lorenz, D., Esser, P., Ommer, B.: High-resolution image synthesis with latent diffusion models (2021)

[38] Saharia, C., Chan, W., Saxena, S., Li, L., Whang, J., Denton, E.L., Ghasemipour, K., Gontijo Lopes, R., Karagol Ayan, B., Salimans, T., et al.: Photorealistic text-to-image diffusion models with deep language understanding. Advances in Neural Information Processing Systems **35**, 36479–36494 (2022)

[39] Schuster, T., Kalyan, A., Polozov, A., Kalai, A.T.: Programming puzzles. In: Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (2021), https://arxiv.org/abs/2106.05784

[40] Shinn, N., Labash, B., Gopinath, A.: Reflexion: an autonomous agent with dynamic memory and self-reflection. arXiv preprint arXiv:2303.11366 (2023)

[41] Sudhakaran, S., González-Duque, M., Glanois, C., Freiberger, M., Najarro, E., Risi, S.: Mariogpt: Open-ended text2level generation through large language models (2023)

[42] Sun, Z., Shen, Y., Zhou, Q., Zhang, H., Chen, Z., Cox, D., Yang, Y., Gan, C.: Principle-driven self-alignment of language models from scratch with minimal human supervision. arXiv preprint arXiv:2305.03047 (2023)

[43] Szerlip, P., Stanley, K.: Indirectly encoded sodarace for artificial life. In: ECAL 2013: The Twelfth European Conference on Artificial Life. pp. 218–225. MIT Press (2013)

[44] Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al.: Llama: Open and efficient foundation language models. arXiv preprint arXiv:2302.13971 (2023)

[45] Vassiliades, V., Chatzilygeroudis, K., Mouret, J.B.: Using centroidal voronoi tessellations to scale up the multidimensional archive of phenotypic elites algorithm. IEEE Transactions on Evolutionary Computation **22**, 623–630 (2016)

[46] Vassiliades, V., Chatzilygeroudis, K., Mouret, J.B.: Using Centroidal Voronoi Tessellations to Scale Up the Multi-dimensional Archive of Phenotypic Elites Algorithm. IEEE Transactions on Evolutionary Computation p. 9 (2017). https://doi.org/10.1109/TEVC.2017.2735550, https://inria.hal.science/hal-01630627

[47] Wang, R., Lehman, J., Rawal, A., Zhi, J., Li, Y., Clune, J., Stanley, K.: Enhanced POET: Open-ended reinforcement learning through unbounded invention of learning challenges and their solutions. In: III, H.D., Singh, A. (eds.) Proceedings of the 37th International Conference on Machine Learning. Proceedings of Machine Learning Research, vol. 119, pp. 9940–9951. PMLR (13–18 Jul 2020), https://proceedings.mlr.press/v119/wang20l.html

[48] Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., et al.: Emergent abilities of large language models. arXiv preprint arXiv:2206.07682 (2022)

[49] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., Zhou, D.: Chain of thought prompting elicits reasoning in large language models. arXiv preprint arXiv:2201.11903 (2022)

[50] White, J., Fu, Q., Hays, S., Sandborn, M., Olea, C., Gilbert, H., Elnashar, A., Spencer-Smith, J., Schmidt, D.C.: A prompt pattern catalog to enhance prompt engineering with chatgpt. arXiv preprint arXiv:2302.11382 (2023)

[51] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Le Scao, T., Gugger, S., Drame, M., Lhoest, Q., Rush, A.M.: Transformers: State-of-the-Art Natural Language Processing. pp. 38–45. Association for Computational Linguistics (Oct 2020), https://www.aclweb.org/anthology/2020.emnlp-demos.6

[52] Xu, C., Sun, Q., Zheng, K., Geng, X., Zhao, P., Feng, J., Tao, C., Jiang, D.: Wizardlm: Empowering large language models to follow complex instructions. arXiv preprint arXiv:2304.12244 (2023)

[53] Young, E.G., Zhu, P., Caraza-Harter, T., Arpaci-Dusseau, A.C., Arpaci-Dusseau, R.H.: The true cost of containing: A gvisor case study. In: Proceedings of the 11th USENIX Conference on Hot Topics in Cloud Computing. p. 16. HotCloud'19, USENIX Association, USA (2019)

[54] Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J.: Large language models are human-level prompt engineers. arXiv preprint arXiv:2211.01910 (2022)

Table 3. Details

batch size	learning rate	weight decay	optimizer
0.5M	1e-5	2e-2	AdamW

A Diff Models

A.1 Hyperparameters

To fine-tune the 350M, 2B, 6B models, we use the same parameters as in Table 3. The training uses a linear warmup of 895 steps out of a total of 5572 steps.

B Sodarace Domain

```python
from openelm.environments.sodaracer.walker import walker_creator


def make_square(wc, x0, y0, x1, y1):
    """Make a square with top left x0,y0 and top right x1,y1."""
    j0 = wc.add_joint(x0, y0)
    j1 = wc.add_joint(x0, y1)
    j2 = wc.add_joint(x1, y1)
    j3 = wc.add_joint(x1, y0)
    return j0, j1, j2, j3


def make_walker():
    wc = walker_creator()

    # the main body is a square
    sides = make_square(wc, 0, 0, 10, 10)
    center = wc.add_joint(5, 5)

    # connect the square with distance muscles
    for k in range(len(sides) - 1):
        wc.add_muscle(sides[k], sides[k + 1])
    wc.add_muscle(sides[3], sides[0])

    # one prong of the square is a distance muscle
    wc.add_muscle(sides[3], center)

    # the other prongs from the center of the square are active
    wc.add_muscle(sides[0], center, 5.0, 0.0)
    wc.add_muscle(sides[1], center, 10.0, 0.0)
    wc.add_muscle(sides[2], center, 2.0, 0.0)

    return wc.get_walker()
```

Listing 1: Sodarace square seed

C Image Generation Domain

```
import math
import numpy as np

# Old version of draw()
# TODO: fix bugs in the code below
def draw():
...

# Fixed version of draw()
def draw():
    """
    Draws a yellow circle with radius 10 in the middle of a 32 by 32 black image.

    Returns:
        np.ndarray: the image
    """
    pic = np.zeros((32, 32, 3))
```

Listing 2: The mutation prompt used to generate new programs for the image generation environment.

```python
import math
import numpy as np

# Fixed version of draw()
def draw():
    """
    Draws a yellow circle with radius 10 in the middle of a 32 by 32 black image.

    Returns:
        np.ndarray: the image
    """
    pic = np.zeros((32, 32, 3))
    pic[16][16] = 1

    # Circle in middle of image
    for x in range(0, 32):
        for y in range(0, 32):
            distance = math.sqrt(math.pow(x - 16, 2) + math.pow(y - 16, 2))
            if distance > 10:
                continue

            pic[x][y] = [255, 255, 0]
    return pic
```

Listing 3: Final evolved image generation program, corresponding to the rightmost image in Figure 4.

D Architext Domain

D.1 A sample JSON document for floor plan design

```
{
  "prompt": "the living room is located in the south west side of the house",
  "layout": {
    "bathroom": [
        ["150", "172"],
        ["150", "128"],
        ["165", "128"],
        ["165", "172"]
    ]
    "bedroom": [
        ["121", "172"],
        ["77", "172"],
        ["77", "143"],
        ["121", "143"]
    ],
    "living_room": [
        ["135", "143"],
        ["77", "143"],
        ["77", "84"],
        ["135", "84"]
    ],
    "kitchen": [
        ["179", "128"],
        ["150", "128"],
        ["150", "84"],
        ["179", "84"]
    ],
    "corridor": [
        ["150", "172"],
        ["121", "172"],
        ["121", "143"],
        ["135", "143"],
        ["135", "84"],
        ["150", "84"]
    ]
  }
}
```

D.2 Architext models

We present some details and demonstrations of using an Architext model ([12], with 162M parameters) as our mutation operator.

We adopted two ways of encoding floor plans into texts:

– The semantic representation introduced in [12]. The following is the initial text from a sample.

```
[prompt] a bedroom is adjacent to the kitchen [layout]
↪  bedroom1: (194,91)(135,91)(135,47)(194,47), living_room:
↪  ...
```

– A JSON format including the prompt and the geometric representation. See Appendix D.1 for a sample JSON document.

In our experiments, the semantic representation applies to Architext models, while the JSON format applies to models capable of JSON document generations.

We use the following two metrics as our diversity metrics:

– Typology: a pair of integers (n, m) for n bedrooms and m bathrooms. It is a categorical metric and we limit n, m to reasonable bounds such as $0 \leq n, m \leq 5$.

– Gross floor area entropy: the entropy of the discrete random variable with values of room areas. More precisely, given a collection of polygons $\{P_1, \cdots, P_n\}$, the gross floor area entropy is defined by the following,

$$-\sum_{i=1}^{n} area(P_i)\, log(area(P_i)),$$

where $area(\cdot)$ is the area of a polygon.

We use the Heat Loss Form Factor (HLFF) as our quality metric. HLFF measures the efficiency of the thermal envelope of a building and is regularly used in Passivhaus buildings. It is the ratio of of surface area that can lose heat (the thermal envelope) to the floor area that gets heated (TFA). To simplify the situation, we assume that all rooms have a uniform height of 2.0. In order to precisely describe our formula of HLFF, given a collection of polygons $\{P_1, \cdots, P_n\}$, we first take the union of all polygons

$$P = \bigcup_{i=1}^{n} P_i.$$

Let $c(P)$ denote the circumference of the polygon P. Then

$$\text{HLFF} = \frac{2 \times area(P) + c(P) \times height}{area(P)}, \tag{1}$$

where $2 \times area(P)$ counts both the floor and the ceiling, and $height = 2.0$ represents our assumption of uniform room heights.

HLFF measures the compactness of a building which in turn is a proxy of energy efficiency, since the more compact a building is, the less insulation will be required for the building to be energy efficient.

Prompt example Architext model is already fine-tuned on semantic representations. Therefore, the prompts are designed to be the initial texts of semantic representations and we perform text completions to generate designs. For the first few initial generations, we fix a list of 58 natural language descriptions, such as

```
the bedroom is adjacent to the living room
```

We sample them uniformly, and wrap it between "prompt" and "layout" tags to make it the initial text of the semantic representation, like the following for the above example,

```
[prompt] the bedroom is adjacent to the living room [layout]
```

Using this, we let the Architext model complete the prompt, and parse the room coordinates out of the generated semantic representation.

For mutations, given a semantic representation such as

```
[prompt] a bedroom is adjacent to the kitchen [layout] bedroom1:
↪  (194,91)(135,91)(135,47)(194,47), living_room:
↪  (121,194)(47,194)(47,91)(106,91)(106,106)(121,106), bathroom1:
↪  (179,121)(135,121)(135,91)(179,91), bedroom2:
↪  (209,165)(135,165)(135,121)(209,121), bathroom2:
↪  (165,209)(135,209)(135,165)(165,165), bedroom3:
↪  (121,238)(47,238)(47,194)(121,194), kitchen:
↪  (135,77)(106,77)(106,18)(135,18), corridor:
↪  (121,209)(121,106)(106,106)(106,77)(135,77)(135,209)
↪  <|endoftext|>
```

we randomly keep the first $1-3$ rooms, keep the name of the next room, and cut off the rest, such as

```
[prompt] a bedroom is adjacent to the kitchen [layout] bedroom1:
↪  (194,91)(135,91)(135,47)(194,47), living_room:
```

The Architext model will complete the prompt and we parse from the result.

Sample generation Figure 7 is a sample 5×5 map from 300 initial random generations and 300 mutations with a batch size of 32 (each generation returns 32 completions as potential designs). The horizontal axis is the typology, and the vertical axis is the gross floor area entropy. Note that this map is far from covering the full domain of all generations, and in fact most of generated designs land outside of the map.

D.3 GPT-3.5 model

We present some details and demonstrations of using OpenAI's GPT-3.5 as our mutation operator.

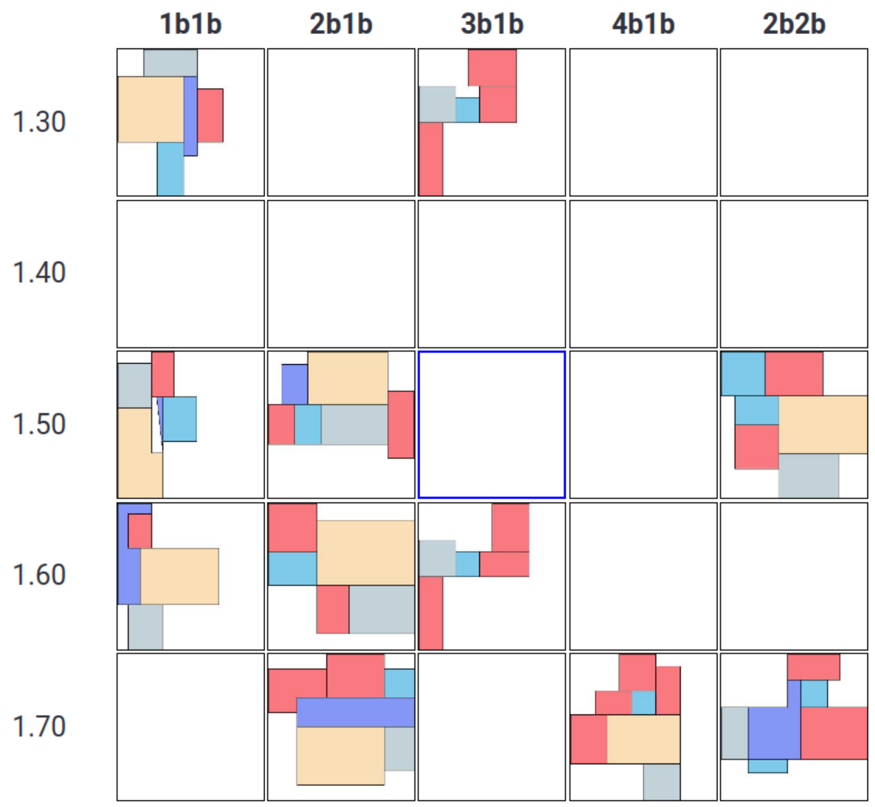


Fig. 7. Architext sample generations

Prompt example In the experiments, we put a JSON example and requirements into a system prompt of GPT-3.5. An example is the following:

```
You are a REST API server receiving prompts describing a floor
↪  plan. You only return JSON documents describing your design.
↪  The format is the following:
1. `prompt`: the original input prompt,
2. `layout`: the room-by-room details of the floor plan in terms
↪  of the coordinates.
An example is the following:
```
```
```

~~~
{'prompt': 'the living room is located in the south west side of
↪  the house', 'layout': {'bathroom': [['150', '172'], ['150',
↪  '128'], ['165', '128'], ['165', '172']], 'bedroom': [['121',
↪  '172'], ['77', '172'], ['77', '143'], ['121', '143']],
↪  'living_room': [['135', '143'], ['77', '143'], ['77', '84'],
↪  ['135', '84']], 'kitchen': [['179', '128'], ['150', '128'],
↪  ['150', '84'], ['179', '84']], 'corridor': [['150', '172'],
↪  ['121', '172'], ['121', '143'], ['135', '143'], ['135', '84'],
↪  ['150', '84']]}}
```
In the `layout` field, each room is represented as a list of
↪  coordinates defining a polygon.
The returned JSON document must satisfy the following
↪  requirements:
1. The prompt is all the info you have. The design detailed in
↪  `layout` follows the prompt as closely as possible.
2. Different rooms cannot overlap.
3. The room names should start with one of 'living_room',
↪  'kitchen', 'bedroom', 'bathroom', 'corridor'.
4. Number of bathroom <= Number of bedroom <= 4.
5. The return must be a valid JSON document.
~~~

To get a new JSON design, we then pass in a user prompt such as

```
the bedroom is adjacent to the living room
```

and parse the result by finding the JSON document and all its keys and values.

Sample generations Figure 8 is the result of a sample generation on a 5×5 map. The horizontal axis is the typology, and the vertical axis is the gross floor area entropy.

The experiment is performed in the following order:

1. We fix 5 original designs (but not included in the map).
2. We sample the 5 original designs uniformly, use the above prompts to perform the chat completions 100 times (initial generations).
3. We then only use the available designs on the map to perform chat completions 200 times (mutation).

We would like to point out that Figure 7 and 8 come as a part of an ongoing project with interactive maps and guided mutations. Figure 9 is an example of the UI prototype.

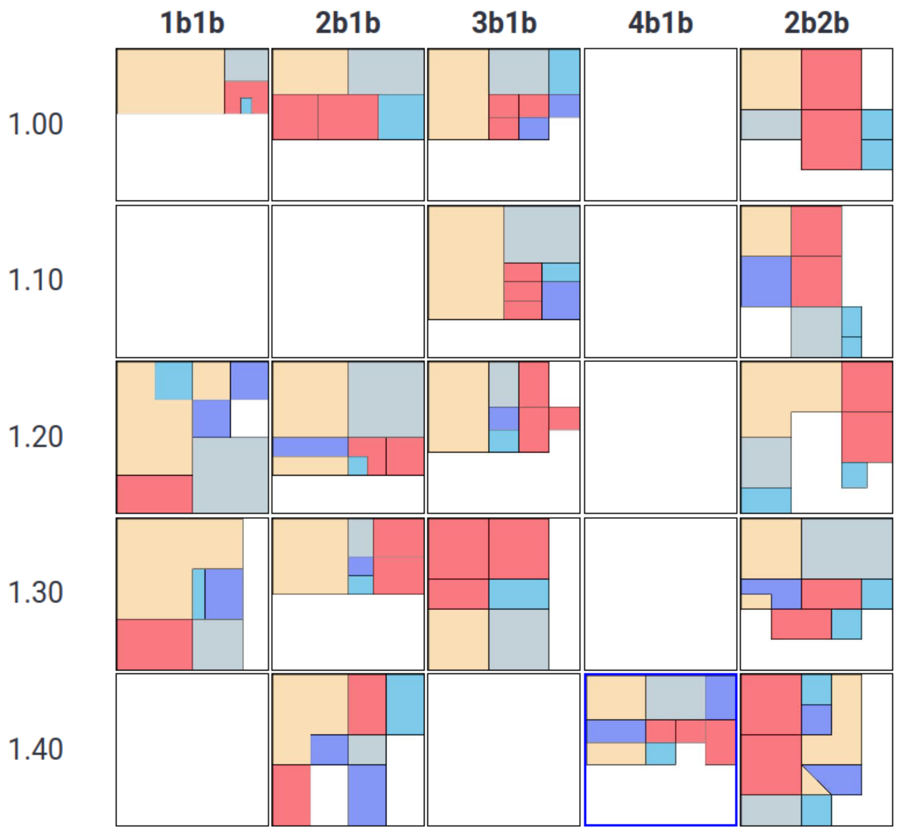
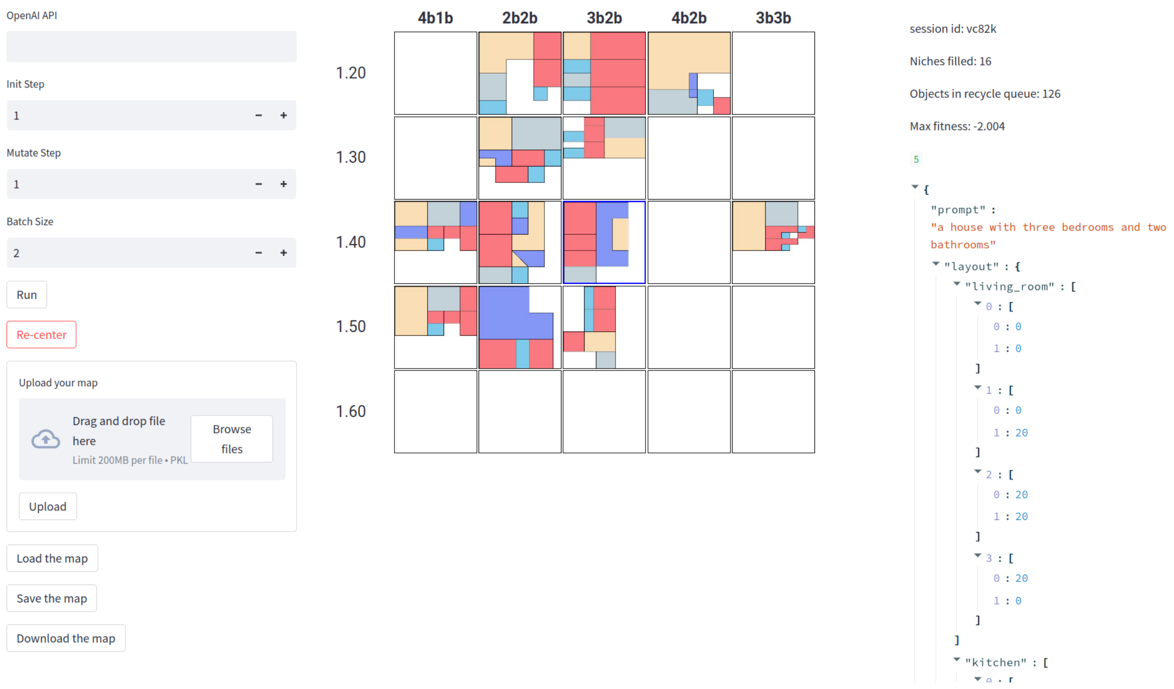


Fig. 8. Architext+GPT3.5+OpenELM sample generations

E Prompt Domain

In this setup, `bugfix_instruction` and `docstring_instruction_str` are filled in with evolved content. `bugfix_instruction` includes the instructions for fixing bugs in the previous code, and `docstring_instruction_str` includes the natural language docstring for the new code. `old_program_str` is filled in with an individual from the image generation environment, and `program_str` is used to evaluate the prompt.



OpenAI API

Init Step
1

Mutate Step
1

Batch Size
2

session id: vc82k

Niches filled: 16

Objects in recycle queue: 126

Max fitness: -2.004

```
5
{
  "prompt" :
  "a house with three bedrooms and two
  bathrooms"
  "layout" : {
    "living_room" : [
      0 : [
        0 : 0
        1 : 0
      ]
      1 : [
        0 : 0
        1 : 20
      ]
      2 : [
        0 : 20
        1 : 20
      ]
      3 : [
        0 : 20
        1 : 0
      ]
    ]
    "kitchen" : [
      0 : [
```

Fig. 9. Architext guided mutation sample UI

```
import math
import numpy as np

# Old version of draw()
# TODO: {bugfix_instruction_str}
def draw():
    {old_program_str}

def draw():
    \"\"\"
    {docstring_instruction_str}

    Returns:
        np.ndarray: the image
    \"\"\"
    pic = np.zeros((32, 32, 3))
    {program_str}
```

Listing 4: Template for evolving over the image generation prompt.

F Programming Puzzles Domain

In `P3Problem`, we prompt in a manner similar to the image generation environment, providing the previous individual along with guiding comments indicating that it needs to be fixed. However, we also prepend the same long prompt used in Schuster et al. [39], which contains 5 example problems and solutions along with docstrings.

In `P3ProbSol`, we augment the long prompt to provide pairs of pairs: (f1_1, g1_1) and (f1_2, g1_2), where _1 is an "original" and _2 is a “new” pair. See Listing F for an example. The problem and solution pair to evolve is appended at the end as (f6_1, g6_1) to prompt the model to generate (f6_2, g6_2).

In the case of the `Salesforce-2B-mono model`, given a text string, the feature matrix has 2560 entries per token and PCA reduces this by about 10-fold while retaining 95% of the variance. Then, we take the max across tokens.

```python
from typing import List

def f1(s: str):
    return "Hello " + s == "Hello world"

def g1():
    """Find a string that when concatenated onto 'Hello ' gives 'Hello world'."""
    return "world"

assert f1(g1())

def f2(s: str):
    return "Hello " + s[::-1] == "Hello world"

def g2():
    """Find a string that when reversed and concatenated onto 'Hello ' gives 'Hello world'."""
    return "world"[::-1]

assert f2(g2())

def f3(x: List[int]):
    return len(x) == 2 and sum(x) == 3

def g3():
    """Find a list of two integers whose sum is 3."""
    return [1, 2]

assert f3(g3())

def f4(s: List[str]):
    return len(set(s)) == 1000 and all(
        (x.count("a") > x.count("b")) and ('b' in x) for x in s)

def g4():
    """Find a list of 1000 distinct strings which each have more 'a's than 'b's and at least one 'b'.
    return ["a"*(i+2)+"b" for i in range(1000)]

assert f4(g4())

def f5(n: int):
    return str(n * n).startswith("123456789")

def g5():
    """Find an integer whose perfect square begins with 123456789 in its decimal representation."""
    return int(int("123456789" + "0"*9) ** 0.5) + 1

assert f5(g5())

def f6(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

# Old version of g6()
# TODO: fix bugs in the code below
...

# Fixed version of g6()
def g6():
    """Find a list of length 10 where the fourth element occurs exactly twice."""
```

Listing 5: Prompt structure for P3Problem. Adapted from [39].

```
from typing import List

...
def f3_1(x: List[int]):
    return len(x) == 2 and sum(x) == 3

def g3_1():
    """Find a list of two integers whose sum is 3."""
    return [1, 2]

assert f3_1(g3_1())

def f3_2(x: List[int]):
    """Changes from f3_1: change sum to 8; add requirement for product to equal 12"""
    return len(x) == 2 and and x[0]+x[1] == 8 and x[0]*x[1] == 12

def g3_2():
    """Find a list of two integers whose sum is 8 and product is 12."""
    return [2, 6]

assert f3_2(g3_2())
...
```

Listing 6: Snippet of the prompt for P3ProbSol. Adapted from [39].

```
'''
Original problem and solution pair for comparison:

def f6(li: List[int]):
    return len(li) == 10 and li.count(li[3]) == 2

def g6():
    """Find a list of length 10 where the fourth element occurs exactly twice."""
    return list(range(10 // 2)) * 2
'''

from typing import List

def f6_2(li: List[int]):
    """Changes from f6_1: all elements must be an integer greater than 3"""
    return len(li) == 10 and li.count(li[3]) == 2 and all(x > 3 for x in li)

def g6_2():
    """Find a list of length 10 where the fourth element occurs exactly twice and
    each integer is greater than 3."""
    return list(range(10 // 2 + 1, 10 + 1)) * 2
```

Listing 7: The result of successfully evolving a problem and solution pair in P3ProbSol.