Random Baselines for Simple Code Problems are Competitive with Code Evolution

Yonatan Gideoni[†*] **Yujin Tang**[§] **Sebastian Risi**[§‖] **Yarin Gal**[†]
[†]OATML, University of Oxford [§]Sakana AI [‖] IT University of Copenhagen

Abstract

Difficult coding problems are often solved by prompting large language models to generate programs and iterate on their code until they find a solution. Many works have proposed ways to guide this iterative process but often do not compare to simpler baselines. Taking nine problems from the AlphaEvolve paper [Novikov et al., 2025] as case studies, we find that randomly sampling programs using an LLM works well, matching AlphaEvolve on two and matching or improving over a strong open source baseline, ShinkaEvolve, on eight. This implies that some improvements may stem not from the LLM-driven program search but due to the manual formulation that makes the problems easily optimizable.

1 Introduction

It is difficult designing computer programs to solve a given task. Programs have precise syntax, long range dependencies, and contain many possible characters, so searching over program space is nontrivial as most randomly sampled strings are invalid. In the last few years many works [Li et al., 2022, Romera-Paredes et al., 2024, Novikov et al., 2025] sample programs using large language models (LLMs), as they can output syntactically valid code [Chen et al., 2021], deal with long range dependencies [Bahdanau et al., 2016], and can generate many different kinds of characters [Sennrich et al., 2015]. In turn, the LLM can be used to "evolve" the code so it better solves a problem, using the LLM as the evolutionary algorithm's mutation operator.

Works like FunSearch [Romera-Paredes et al., 2024] and AlphaEvolve [Novikov et al., 2025] discovered new mathematical bounds and algorithms using this process. In spite of these problems' complex formulations, some of them are functionally simple, having relatively low dimensional input spaces – consisting of $10^0 - 10^4$ numbers – with their optimal solutions being straightforward programs for black-box numerical optimization (see Listing 1).[2] This begs the question, how well would much simpler methods do?

As a first step towards answering this question, we run random search baselines over nine problems from the AlphaEvolve paper. As it is unclear how those solutions were found and given how much resources, we compare a similar open source sample efficient system, ShinkaEvolve Lange et al. [2025], to randomly sampling programs from an LLM. On two of the nine problems random search achieves the same performance ceiling as AlphaEvolve while on eight of the nine it matches or outperforms ShinkaEvolve. Given the same API budget, random search has a $> 50\%$ chance of outperforming ShinkaEvolve on five of the nine problems. Thus, for easily verifiable problems with short programs as answers, the hard part may be formulating the problems so they are easily optimizable – which is still done by hand – and not the LLM-driven optimization itself.[3]

[*]Work done during an internship at Sakana AI. Email: yg@robots.ox.ac.uk

[2]This is seen in AlphaEvolve's limited shared code and open source replications like Sharma [2025], Lange et al. [2025].

[3]Code will be open sourced in the paper's final version and meanwhile will be shared upon request.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: Deep Learning for Code.

```python
def pack_circles() -> Tuple[np.ndarray, np.ndarray]:
    ...
    # Try to arrange the circles in a grid-like structure
    initial_centers = np.array([[0.2, 0.2], [0.8, 0.2], [0.2, 0.8], [0.8, 0.8],
    ...
    # Define bounds for the optimizer
    bounds = [(0, 1) for _ in range(52)] + [(0, 0.5) for _ in range(26)]
    ...
    result = minimize(
        calculate_objective,
        x0,
        method='SLSQP',
        bounds=bounds,
        constraints=calculate_constraints(x0),
        options={'maxiter': 2000, 'ftol': 1e-6}
    )
    ...
    return centers, radii
```

Listing 1: Code snippets from the best circle packing solution found by a random baseline here. The function is fairly straightforward, using simple black-box numerical optimization.

2 Problems

Novikov et al. [2025] demonstrated using LLM driven code evolution to solve many tasks, among them finding bounds for mathematical problems. We focus on these kinds of problems as they are relatively simple, requiring relatively short single-file programs and quick CPU based evaluation. These problems are subdivided into those belonging to analysis, combinatorics, or geometry, with us using three, two, and four problems from each category respectively.[4] Brief summaries of the problems and their bounds are in Table 1, with longer explanations in Novikov et al. [2025]'s Appendix.

3 Random Baselines

There are largely two ways to randomly sample solutions – at the input level and at the program level, with only the latter requiring an LLM. Searching directly at the input level, where the problem's parameters are sampled from some distribution, can work for low dimensional problems but struggles scaling for cases with even tens of dimensions. We show this in Appendix A over a subset of three problems. Random search over the inputs matches AlphaEvolve on the uncertainty inequality, having a 3-dimensional input, but gets subpar performance for the others.

When searching over program space how much domain knowledge should be used? While on the one hand it may guide the model towards better solutions it can also make it get stuck on suboptimal setups. Moreover, these biases confound how well a method works with how the model is guided. Thus, we opt to give relatively little domain knowledge in the model's prompts, specifying only the problem's broad structure and some evaluation functions. An example prompt, for the second autocorrelation inequality, is given in Appendix C.

This relatively little domain knowledge also facilitates a fair comparison with different methods that get more than just prompts as inputs. ShinkaEvolve and other code evolution systems [Novikov et al., 2025, Sharma, 2025] iterate over an initial program, starting their search with some biases. For example, Novikov et al. [2025], Lange et al. [2025] start their circle packing program with a function that, given a configuration of circle centers, finds their maximal radii. When asking Gemini 2.5 Pro to generate ten circle packing solutions with a prompt with minimal biases it never includes such a function in its solution. Therefore, ShinkaEvolve is always initialized from a trivial initial program with essentially no domain knowledge other than the functions the random sampling baseline also has access to.

[4] Combinatorics has only two problems, hence the imbalance.

Problem	Input size	Pre-AE Bound	AE Bound	AE Appendix
First autocorrelation inequality (↓)	Unbounded, step function heights	1.5098	1.5053	B.1
Second autocorrelation inequality (↑)	Unbounded, step function heights	0.88922	0.8962	B.2
Uncertainty inequality (↓)	3 coefficients of a Hermite polynomial	0.35229	0.35210	B.4
Erdős' minimum overlap (↓)	Unbounded, step function heights	0.380927	0.380924	B.5
Sums vs. differences of finite sets (↑)	Unbounded, set $U \subset \mathbb{Z}_{\geq 0}$ fulfilling some properties	1.14465	1.1584	B.6
Max–min distance ratio for 16 2D points (↓)	32 coordinates (16×2)	12.890	12.88927	B.8
Heilbronn triangles $n = 11$, (↑)	22 coordinates (11×2) in a unit-area triangle	0.036	0.0365	B.9
Kissing number in 11D (↑)	Largest number of 11D sphere centers all tangent to a common sphere	592	593	B.11
Circle packing (↑)	78 – 26 center coordinates (26×2) and 26 radii	2.634	2.63586	B.12

Table 1: Bounds of AlphaEvolve (AE) problems studied here, divided into analysis (top), combinatorics (middle), and geometry (bottom). The arrow next to the problem name indicates whether it is an upper bound, so lower results are tighter and hence better (↓), or a lower bound so higher is tighter and thus better (↑). All numbers are from Novikov et al. [2025]. "Pre-AE" are the best bounds from before Novikov et al. [2025].

Concretely, for the randomly sampled programs we use Gemini 2.5 Pro, sampling with a temperature of 0.8, a top-p sampling cutoff of 0.95, a thinking budget of 1024 tokens, and letting each solution run for at most 5 minutes. These settings were not thoroughly ablated and chosen as they seemed like sensible defaults. During development we observed a general, intuitive trend of more thinking tokens and a longer execution time giving better results, while making API calls more expensive and experiments take longer. Given a prompt describing a problem we ask the LLM to output entire programs and extract ```` ```python ... ``` ```` from its completions. 2000 solutions are sampled for each problem, requiring 25-50$ worth of API calls per problem.

To make the comparison fair we have ShinkaEvolve uses only LLMs from the Gemini family – specifically 2.5 Pro, Flash, and Flash Lite – and adopt their circle packing hyperparameters, changing the system prompts and initial programs per problem. Each problem is run for 200 evolution generations, making each run cost 12-18$, where their reference circle packing run costs about 12$.

4 Results

To see how well random search would perform given the same API budget as ShinkaEvolve, we calculate the probability random search would have matched or outperformed it for each problem, $P_{\text{RS}\geq\text{Shinka}}$ given the same settings. This is done by (conservatively) assuming a ShinkaEvolve run costs 12$, seeing how many random programs can be sampled for that price, and calculating the probability sampling that many programs yields one which is better than Shinka's optimal solution, which amounts to calculating a pass@k. Many other metrics can be compared – wall clock time,

number of output tokens, etc. – but are harder to do so fairly as they depend more on implementation details.[5]

As Table 2 shows, random search outperformed ShinkaEvolve on most problems, both in general and when considering its probability of outperforming Shinka given the same budget. This is especially surprising for problems with variable, typically unbounded, inputs, which are five out of the nine problems. Intuitively, for these cases a method that builds on previous solutions would perform better, so why did Shinka not here? One option is that being incremental biases it towards more common, worse solutions, where it can be easier directly sampling a low probability good solution instead.

Problem		AE	Random Search	ShinkaEvolve	$P_{\text{RS}\geq\text{Shinka}}$
First autocorr. ineq.	(↓)	**1.5053**	*1.529*	1.541	95.2%
Second autocorr. ineq.	(↑)	**0.8962**	*0.8739*	0.868543	41.7%
Uncertainty ineq.	(↓)	**0.3521**	**0.3521**	**0.3521**	100%
Erdős' min. overlap	(↓)	**0.3809**	*0.3811*	0.3813	44.2%
Sums/differences of sets	(↑)	**1.1584**	*1.124*	1.110	100%
Max–min dist. ratio	(↓)	*12.88926*	**12.88923**	12.99	100%
Heilbronn triangles	(↑)	**0.0365**	0.0334	*0.0356*	0%
Kissing number in 11D	(↑)	**593**	*438*	405	43.4%
Circle packing	(↑)	**2.635**	*2.632*	2.621	100%

Table 2: ShinkaEvolve and random search baselines. Best results are bolded, second best are italicized. Random search matches or exceeds AlphaEvolve on two of the nine problems and ShinkaEvolve on eight of the nine. Arrows denote whether higher or lower is better.

Why did ShinkaEvolve find a subpar circle packing, although Lange et al. [2025] got better results than AlphaEvolve? This is likely due to having both a system prompt and initial program with much less domain knowledge. In Appendix A we see that randomly sampling a thousand programs when given more domain knowledge results in a similarly good packing configuration as well.

Why do both baselines generally underperform AlphaEvolve? Other than likely using far fewer resources,[6] it is unclear how open-ended AlphaEvolve is, e.g. to what extent are its prompts especially designed per problem? Without an open source reproduction that matches its performance it is hard to do more than speculate.

5 Discussion

Code evolution systems are often presented as intended for advancing scientific discovery [Novikov et al., 2025, Lange et al., 2025], where it is fair to use domain knowledge if it helps solve a problem. Meanwhile, common machine learning wisdom argues that using algorithms with more human knowledge is detrimental, yielding short term gains that are surpassed by more general long term improvements [Sutton, 2019]. Interestingly, given the same budget and domain knowledge, a simple random search baseline seems competitive with code evolution on several mathematical problems.

This has several implications. First, when developing these systems, what should one focus on? Is it wall clock time, or getting good performance within a given budget, or something else entirely? Typically these constraints are not well defined, but they are important for the field to progress. Simple baselines performing well might show that current systems are insufficiently sample efficient, or prioritize the wrong metrics, or something else entirely.

More fundamentally, what do we care about solving? In mathematics improving a bound is typically interesting only if it yields some deeper insights [Tao, 2007], whereas in machine learning getting better performance on a problem is practically useful even if it is due to mundane reasons. This is generally the difference between natural and engineering sciences, so systems for (natural) scientific discovery should focus on yielding insights more than improving bounds and performance. Regardless of the science, all good results are either useful or interesting, and in the best cases both. If what

[5]For example, given enough CPUs random search can be done very quickly while many code evolution systems have sequential components.

[6]Novikov et al. [2025] mention using "thousands of LLM samples" per problem.

matters for scientific discovery is finding good problem formulations, with the optimization afterwards being easy, how can we teach a model to better formulate problems?

A nice example of how formulations are important relative to the optimization process is in Appendix B.4 of Novikov et al. [2025]. After publishing the paper the AlphaEvolve authors were told of a better formulation of the uncertainty inequality problem, which had a published lower bound of 0.3284 instead of the previous one of 0.3523. This allowed them to find a new bound of 0.3216 instead of 0.3521. Although the optimization improved both bounds, the larger improvements can from a better underlying structure.

Still, these results should be taken with caution. Code evolution systems have evidently been useful, with AlphaEvolve finding better tensor factorizations and ShinkaEvolve being used to win a coding competition, yielding deeper insights on the way [Akiba, 2025]. Random baselines, especially non-iterative ones, likely have their limitations when applied to more difficult highly structured problems, where a good sample efficiency is important. These problems are harder to work with but might be a better focus than proof of concepts like circle packing. Developing better benchmarks and understanding when and how code evolution's value lies is important for future work.

Acknowledgments and Disclosure of Funding

We would like to thank numerous members of the Sakana AI research team for many fruitful discussions. Thanks also to Edan Toledo and Dulhan Jayalath for nice feedback. Yonatan is funded by the Rhodes Trust and the AIMS EPSRC CDT (grant no. EP/S024050/1).

References

Takuya Akiba. ShinkaEvolve in Action: How a Human–AI Partnership Conquered a Coding Challenge. `https://sakana.ai/icfp-2025/`, October 2025. Sakana AI blog.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate, 2016. URL `https://arxiv.org/abs/1409.0473`.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde De Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Fuchang Gao and Lixing Han. Implementing the nelder-mead simplex algorithm with adaptive parameters. *Computational Optimization and Applications*, 51(1):259–277, 2012.

Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. Shinkaevolve: Towards open-ended and sample-efficient program evolution. *arXiv preprint arXiv:2509.19349*, 2025.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097, 2022.

Alexander Novikov, Ngân Vũ, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco JR Ruiz, Abbas Mehrabian, et al. Alphaevolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint arXiv:2506.13131*, 2025.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015.

Asankhaya Sharma. Openevolve: an open-source evolutionary coding agent, 2025. URL `https://github.com/codelion/openevolve`.

Richard Sutton. The bitter lesson. *Incomplete Ideas (blog)*, 13(1):38, 2019.

Terence Tao. What is good mathematics? *Bulletin of the American Mathematical Society*, 44(4): 623–634, 2007.

A Input Level Random Search

We compare two kinds of input random sampling to LLM based program sampling. The simplest method is **direct random sampling**, where the problem parameters are randomly sampled from a given distribution, e.g. uniformly over $[0, 1]$. If the function is well behaved then **random sampling with numerical optimization** should perform better, where the initial guess is sampled like before but then optimized using a black-box optimizer, here being Nelder-Mead [Gao and Han, 2012]. Most general is **LLM-based random sampling** where an LLM is prompted to generate a program that produces the optimal parameters.

Direct and numerical optimization based random sampling are run for each problem for 6 hours over 8 CPU cores. This results in a tradeoff between the number of sampled solutions and how well each is optimized, as the optimization based method samples fewer initial points but spends wall clock time on optimizing them. For LLM based sampling we generate a thousand programs for each problem. To see if a better LLM produces better programs this is done twice, once using Gemini 2.0 Flash Lite and again with Gemini 2.5 Pro. For both numerical optimization and LLM based sampling we limit each program to run for at most 60 seconds.

For the input level sampling, for Erdős' minimum overlap and circle packing problems the inputs are sampled uniformly from $[0, 1]$. For the uncertainty inequality the three inputs are sampled log-uniformly from $[10^{-2}, 10^{2}]$, $[10^{-4}, 10^{0}]$, $[10^{-6}, 10^{-2}]$, with the rough orders of magnitude chosen based on the pre-AlphaEvolve optimal solution's coefficients.

Erdős' minimum overlap problem can have any arbitrary step function as its input. For a fair comparison with the AlphaEvolve solution we use 95 steps for the direct and numerical optimization baselines while for the LLM based sampler we accept any number but prompt the model to use 95. This may make the problem either easier or harder, as on the one hand it is less open ended but on the other it is confined to a smaller search space.

For circle packing naïvely sampling centers and radii would most of the time yield invalid solutions. Thus, we use some domain knowledge and formulate the problem as a 52-dimensional optimization problem, where given the 26 centers the optimal radii are inferred. This can be done by solving a linear programming problem, which is quick in practice. Details are in Appendix B. This makes sampling valid solutions feasible but may give these baselines an advantage.[7] For LLM based sampling the model has no access to the linear program.

However, as in this case the circle packing input-level sampling has additional domain knowledge in the form of the linear program we give the LLMs access to the circle packing initial program helper function from [Novikov et al., 2025] as a possible helper. This allows random search to get a better packing than AlphaEvolve, whereas in Table 2 without this additional knowledge it underperformed. This is an example of domain knowledge being the difference between getting state of the art and subpar performance.

As seen in Table 3, direct input sampling underperforms sampling programs for the minimum overlap and circle packing, whereas for the very low dimensional uncertainty inequality they all perform similarly.

B Circle Packing Linear Program

Given n circle centers x_i, y_i we wish to find their radii r_i such that $\sum_i r_i$ is maximized while all the circles are in the unit square and do not overlap. Assuming the circle centers are valid, the no-overlap constraint for circles i, j is $d_{ij} \leq r_i + r_j$ where $d_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$. Circle i's maximum radius without exiting the square is $u_i := \min(x_i, y_i, 1 - x_i, 1 - y_i)$, yielding the

[7] It is worth noting that LLMs can easily reproduce the linear programming formulation when prompted to do so.

Problem	AlphaEvolve	Direct Sampling	Numerical Optimization	LLM Flash Lite	LLM Pro
Uncertainty inequality (↓)	**0.35210**	0.35216	**0.35210**	0.35213	**0.35210**
Erdős' minimum overlap (↓)	**0.38092**	0.44855	0.39662	0.38242	0.38233
Circle packing (↑)	2.63586	1.78428	2.20161	2.52451	**2.63598**

Table 3: Best bounds found by AlphaEvolve and random search baselines. On two out of the three problems random search matched or slightly exceeded AlphaEvolve's bounds.

constraint $r_i \leq u_i$. This also allows pruning the overlap constraints, as if $d_{ij} \geq u_i + u_j$ then the inequality over the radii is always fulfilled. All together this yields a linear program with $O(n^2)$ constraints in the radii, which can be efficiently solved using a variety of tools.

C Second Autocorrelation Inequality Prompt

The `compute_lower_bound` function is taken from AlphaEvolve's validation script. `${max_execution_time}` is replaced with the time limit per problem, which in practice was 300 seconds (5 minutes) for the programs in Table 2 and 60 seconds for those in Appendix A.

````
You are an expert programmer specialising in numerical optimisation. Implement a
↪  Python function with the exact signature:

def find_step_heights() -> np.ndarray:

Where the goal is to find step function heights that will maximize the lower bound
↪  on the smallest constant C for which $$ \|f*f\|_2^2 \leq C \|f*f\|_1
↪  \|f*f\|_\infty$$, $f$ being a nonnegative function supported on $[-1/4, 1/4]$.
↪  The returned np array should represent the heights of this step function,
↪  yielding a constant $K<C$. We wish to maximize $K$.

You can use this predefined helper function without redefining it:
```
def compute_lower_bound(step_heights: np.ndarray) -> float:
    convolution = np.convolve(step_heights, step_heights)

    # Calculate the 2-norm squared: ||f*f||_2^2
    num_points = len(convolution)
    x_points = np.linspace(-0.5, 0.5, num_points + 2)
    x_intervals = np.diff(x_points)  # Width of each interval
    y_points = np.concatenate(([0], convolution, [0]))
    l2_norm_squared = 0.0
    for i in range(len(convolution) + 1):  # Iterate through intervals
        y1 = y_points[i]
        y2 = y_points[i + 1]
        h = x_intervals[i]
        # Integral of (mx + c)^2 = h/3 * (y1^2 + y1*y2 + y2^2) where m = (y2-y1)/h,
        ↪  c = y1 - m*x1, interval is [x1, x2], y1 = mx1+c, y2=mx2+c
        interval_l2_squared = (h / 3) * (y1 ** 2 + y1 * y2 + y2 ** 2)
        l2_norm_squared += interval_l2_squared

    # Calculate the 1-norm: ||f*f||_1
    norm_1 = np.sum(np.abs(convolution)) / (len(convolution) + 1)

    # Calculate the infinity-norm: ||f*f||_inf
    norm_inf = np.max(np.abs(convolution))
    return l2_norm_squared / (norm_1 * norm_inf)
```

Note that all steps should be non-negative. You can have any number of steps in
↪  your step function and up to ${max_execution_time} seconds for your solution
↪  to run. The returned value must be a 1-D NumPy array.
````