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

Sophisticated LLM-based search pipelines have been proposed for AI for scientific discovery, but is their complexity necessary? As a case study, we test how well two kinds of LLM-based random baselines – IID random sampling of programs and a sequential random search – do on nine problems from the AlphaEvolve paper [Novikov et al., 2025], compared to both AlphaEvolve and a strong open-source baseline, ShinkaEvolve. We find that random search works well, with the sequential baseline matching AlphaEvolve on 4/9 problems and matching or improving over ShinkaEvolve, using similar resources, on 7/9. This implies that some improvements may stem not from the LLM-driven program search but from the manual formulation that makes the problems easily optimizable.

1 Introduction

In AI for scientific discovery large language models (LLMs) are often used to search over a complicated space for an object with some properties. For example, Novikov et al. [2025] define various mathematical bounds using Python programs, so a better bound can be found by searching over program space. Their pipeline, AlphaEvolve, found new best bounds for a variety of problems by using an LLM to "evolve" some code, acting as an evolutionary search's mutation operator.

Many works in AI4Science propose complex pipelines and showcase their resulting scientific discoveries, e.g. Lu et al. [2024], Gottweis et al. [2025], Novikov et al. [2025], Lange et al. [2025], Mitchener et al. [2025]. Although this shows how discoveries can be found, often the search method's efficacy isn't tested. Moreover, in spite of these problems' complex formulations, some of them are functionally simple. Many problems solved by AlphaEvolve have 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 search methods do?

As a first step towards answering this, we test how well random search (RS) baselines do on nine problems from the AlphaEvolve paper. As it is unclear how those solutions were found, and given how many resources, we compare to a similar open-source sample-efficient pipeline, ShinkaEvolve [Lange et al., 2025]. Testing two variants of random search – IID random sampling and sequential random search – shows that they perform well, matching or exceeding ShinkaEvolve on 4/9 and 7/9 problems respectively. When API budget-matched, IID and sequential RS have a probability of matching or exceeding code evolution of 44% and 76% respectively. Thus, at least 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 search itself.

[*]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].

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 diverse problems, among them finding mathematical bounds. We focus on these kinds of problems as they are relatively simple, requiring 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.[3] Brief summaries of the problems and their bounds are in Table 2 in Appendix A, with longer explanations in Novikov et al. [2025]'s Appendix.

Each problem defines a function that converts a list of numbers (perhaps with some additional structure) into a bound. For example, one problem is finding the maximum sum of radii of 26 circles in a unit square. Given a list of numbers representing the radii and center locations, then, assuming the circle packing is valid, the bound is the sum of numbers representing the radii. This setup is illustrated in Figure 1a. Code evolution pipelines find programs that generate these numbers.

3 Random Baselines

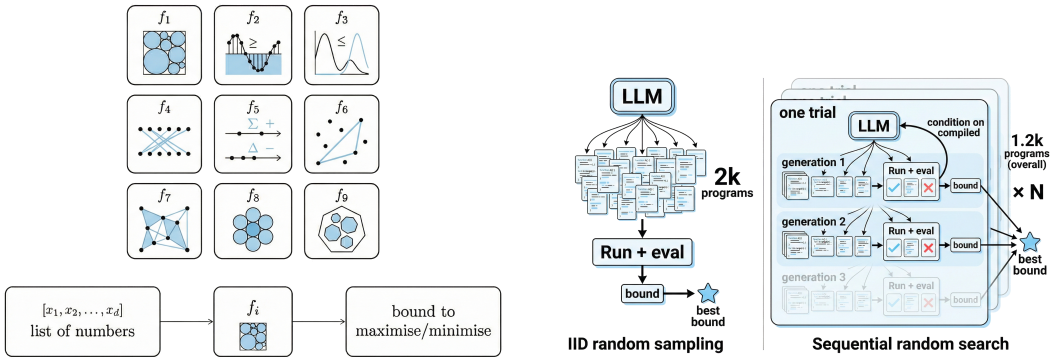
We test two simple ways of searching over programs – IID random sampling and sequential random search. IID RS samples many programs given some prompt defining the problem, evaluates them, and selects the one with the best bound. Sequential RS initially generates programs like IID RS, but then in its following generations[4] includes three randomly selected successful programs in its prompt for the LLM to improve on. "Successful" simply means they returned a valid output, where the selection is otherwise completely random. This is done several times, each time constituting a "trial", after which the best bound is selected among all programs across all generations and trials. Figure 1b illustrates both baselines.

In addition to searching over programs, random search can be done over a different space, like the level of a problem's input. 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. This is shown in Appendix B over a subset of three problems.

For these LLM-based search methods, should domain knowledge be used? While on the one hand it could guide the model towards better solutions it can also lead it down suboptimal trajectories. Moreover, biasing the search confounds how well a method works with how the LLM is guided. Guidance can be desirable when solving a problem, but not necessarily when benchmarking a setup. Thus, to purely test the search methods in and of themselves, we opt to give minimal domain

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

[4] In the evolutionary algorithms sense of the word.



(a) (b)

Figure 1: **(a)** Each problem effectively defines a function that gets as input a list of numbers, possibly with some extra structure, and outputs a mathematical bound that should be maximized or minimized. **(b)** The two baselines, (left) randomly IID sampling a set of programs from an LLM and picking the best one and (right) generating a set of programs, evaluating them, and generating a new set conditioned on some of those that successfully ran.

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 D.

This relatively little domain knowledge also facilitates a fair comparison with methods that get more than just prompts as inputs. ShinkaEvolve and other code evolution systems (e.g. Novikov et al. [2025], Sharma [2025]) iterate over an initial program, starting their search with some guidance. For example, Novikov et al. [2025], Lange et al. [2025], Sharma [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 output. Thus, we choose to always initialize ShinkaEvolve from a trivial initial program with essentially no domain knowledge beyond the functions the RS baselines can access.

Regarding the setup, for the baselines 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 let each program evaluation 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 taking 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 for IID RS and 1200 for sequential RS, using 6 trials with 10 generations and 20 programs per generation. Each baseline requires $25–50 worth of API calls per problem.

To make the comparison fair, we restrict ShinkaEvolve to LLMs from the Gemini family – specifically 2.5 Pro, Flash, and Flash Lite. Hyperparameters are based on their circle packing setup and slightly tuned,[5] while also changing the system prompts and initial programs per problem. Each problem is run until it uses up $20 of API budget, typically yielding 500-800 generations.[6] For reference, the circle packing run in Lange et al. [2025] costs about $12 and ran for 150 generations.

4 Results

To see which search method works best there are two basic questions that should be answered, pertaining to a method's performance and efficiency. First, for a set problem which search method

[5]Most importantly, if Shinka is given unlimited thinking tokens (as it by default is) then it can be very expensive to run for many generations and, given a limited budget, isn't competitive with the RS baselines. Thus, we limit Shinka as well to 1024 thinking tokens per query.

[6]In many cases it is unclear whether using a larger budget would lead to significant improvements as after enough generations ShinkaEvolve seems to plateau.

performs best, given all methods have a reasonable budget?[7] Second, given the same budget, what is the probability one method outperforms another? RS baselines are found to be competitive with code evolution in both regards.

Table 1 shows how well each method performed across all problems. IID RS matches or exceeds Shinka on 4/9 problems, while sequential RS matches or exceeds it on 7/9. Interestingly, sequential RS matches AlphaEvolve on 4/9 problems, despite the latter using an unspecified but potentially much larger budget.

Problem		AE	IID RS	Sequential RS	ShinkaEvolve
First autocorr. ineq.	(↓)	**1.5053**	1.529	*1.519*	1.522
Second autocorr. ineq.	(↑)	**0.8962**	0.8739	0.8795	*0.8955*
Uncertainty ineq.	(↓)	**0.3521**	**0.3521**	**0.3521**	**0.3521**
Erdős' min. overlap	(↓)	**0.3809**	*0.3811*	0.3812	**0.3809**
Sums/differences of sets	(↑)	**1.1584**	*1.1237*	1.1216	1.1095
Max–min dist. ratio	(↓)	*12.88926*	**12.88923**	**12.88923**	**12.88923**
Heilbronn triangles	(↑)	**0.0365**	0.0334	**0.0365**	*0.0356*
Kissing number in 11D	(↑)	**593**	*438*	*438*	402
Circle packing	(↑)	*2.63586*	2.632	**2.63598**	**2.63598**
# problems $\geq$ Shinka		7/9	4/9	7/9	-
# problems $\geq$ AE		-	2/9	4/9	4/9

Table 1: Results for code evolution and random search baselines. Best results are bolded, second best are italicized. Arrows denote whether higher or lower is better. # problems $\geq M$ means the number of problems for which a method matches or exceeds method M.

To see how well the baselines would perform given the same budget as ShinkaEvolve, we calculate the probability RS would have matched or outperformed it for each problem, $P_{\text{RS}\geq\text{Shinka}}$. This is done using a bootstrap estimate over a single pre-existing Shinka run and a set of IID RS samples and sequential RS trials, with details in Appendix E.[8] "Budget" can refer to various metrics, like the API budget or number of evaluated programs, with the main bottleneck defining what's important. As programs are relatively quick to evaluate, here the API budget is the limiting factor. To see how methods compare regarding their sample efficiency, a similar comparison that reaches a similar conclusion is given in Appendix G.

Figure 2 shows the probability of IID and sequential RS matching or exceeding Shinka, averaged over all problems. Shinka has a longer warmup phase than the RS baselines due to various reasons, e.g. sometimes using code diffs instead of generating full files, hence why it is initially much worse than both baselines. After somewhat plateauing at higher budgets IID RS does moderately well while sequential RS decently outperforms code evolution.[9]

There are a few possible reasons why sequential RS outperforms code evolution. Empirically, code evolution suffers from "code bloat", where the number of lines in a program gradually increases, potentially leading to inefficient edits or making the search struggle to go down different paths. This phenomenon is known in classical genetic programming literature [Langdon and Poli, 1997], but might be detrimental here.

Note that neither the RS baselines nor ShinkaEvolve were extensively hyperparameter tuned. The baselines have very few hyperparameters and were tested to be generally operational while ShinkaEvolve was slightly tweaked relative to some default settings, see Appendix H for details. This is

[7]The reasonable budget constraint is important. Given infinite samples, all methods that give a nonzero probability to all programs, as LLM sampling based methods do, would eventually reach the same performance ceiling.

[8]Only a single Shinka run is used per problem as these experiments are very expensive – running Shinka and the two RS baselines costs >$50 per problem. The RS baselines are easier to slightly oversample as they come in smaller discrete chunks than a full Shinka run, being more regular samples for IID RS and more trials for sequential RS.

[9]It is important to note that as the probability of matching or exceeding a result is measured running a method vs itself many times would get a probability of >50%. This is due to ties, e.g. two methods converging to the same bound or reaching a problem's performance ceiling.

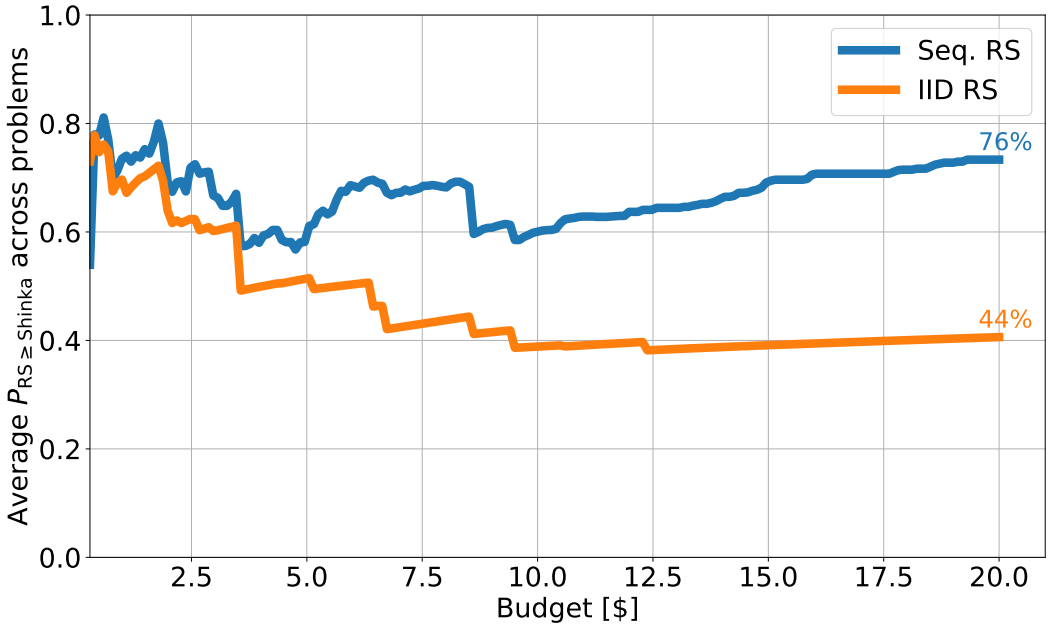


Figure 2: Average probability of matching or exceeding ShinkaEvolve over the 9 problems for the different baselines. Numbers on the right are the probabilities at the max budget of $20.

important as for budget limited search methods the hyperparameter tuning can be much more expensive than the final search, so methods would typically be used out of the box. Moreover, any hyperparameter tuning should be included as part of the search, as a solution found while tuning is still valid. However, this is often not discussed and complicates the resulting analysis. In practice, akin to typical learning setups, some problems should be used for finding hyperparameters (as a validation set) while others are reserved for benchmarking.

Why did ShinkaEvolve find a subpar circle packing here, while Lange et al. [2025] got better results with a lower budget? This is likely due to Shinka here using both a system prompt and initial program with less domain knowledge. Domain knowledge is likely beneficial to any method, not just code evolution – sampling a thousand programs using IID RS with a prompt with more domain knowledge results in a similarly good packing configuration to that found by Lange et al. [2025], as shown in Appendix B.

While sequential RS matches AlphaEvolve on a few problems, it is hard drawing a fair comparison as AlphaEvolve likely uses more resources[10] and potentially more domain knowledge. Still, a simple baseline matching a more complicated system shows that the search is arguably quite simple, at least for these kinds of problems.

5 Discussion

Here we showed that two kinds of random search baselines, IID random sampling and sequential random search, are competitive with code evolution. Thus, for simple problems like those studied here, the main challenge might not be finding sufficiently powerful search methods but rather designing easily optimizable problem formulations, which is still done by human experts.

A nice example of how formulations are potentially more important than 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

[10] Novikov et al. [2025] mention using "thousands of LLM samples" per problem, which could translate to thousands of generations. For reference, here ShinkaEvolve used <800 per problem.

instead of 0.3521. Although the optimization tightened both bounds, the significant improvements came from a better formulation.

It would be interesting to test simple search methods on harder tasks, like the matrix multiplication problem from Novikov et al. [2025] or the mixture of experts loss optimization in Lange et al. [2025]. While the RS baselines seem competitive with code evolution under some constraints, e.g. with a given budget or max number of evaluated programs, this may differ for harder problems with different limiting factors.

More fundamentally, it is worth asking, 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 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?

Given a good scientific problem, it is important to decouple the discovery's importance from that of the method that led to it. Often these are conflated, and focusing on each can lead to very different kinds of research. For example, if one focuses on achieving a specific scientific discovery, then as much domain knowledge as desired can be used, whereas if one proposes a method then it should be compared to others given similar resources and knowledge. Long term, developing good search pipelines requires finding sufficiently difficult problems and comparing them to simpler methods so the new method's efficacy is clear. While doing so it is important to not unintentionally leak domain knowledge into the search through hyperparameter optimizations, which could make the search better than it would be on a new problem. It will be interesting in the future finding harder sets of problems and building on insights from simple baselines to develop better search methods.

Acknowledgments and Disclosure of Funding

We would like to thank numerous members of the Sakana AI research team for many fruitful discussions, and especially for Robert Lange for helping with the ShinkaEvolve baselines. Thanks also to Edan Toledo and Dulhan Jayalath for nice feedback, and to Noya Gideoni for proofreading an earlier draft. Yonatan is funded by the Rhodes Trust and the AIMS EPSRC CDT (grant no. EP/S024050/1).

References

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.

Juraj Gottweis, Wei-Hung Weng, Alexander Daryin, Tao Tu, Anil Palepu, Petar Sirkovic, Artiom Myaskovsky, Felix Weissenberger, Keran Rong, Ryutaro Tanno, et al. Towards an ai co-scientist. *arXiv preprint arXiv:2502.18864*, 2025.

William B Langdon and Riccardo Poli. Fitness causes bloat. In *Soft Computing in Engineering Design and Manufacturing*, pages 13–22. Springer, 1997.

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

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob Foerster, Jeff Clune, and David Ha. The ai scientist: Towards fully automated open-ended scientific discovery. *arXiv preprint arXiv:2408.06292*, 2024.

Ludovico Mitchener, Angela Yiu, Benjamin Chang, Mathieu Bourdenx, Tyler Nadolski, Arvis Sulovari, Eric C Landsness, Daniel L Barabasi, Siddharth Narayanan, Nicky Evans, et al. Kosmos: An ai scientist for autonomous discovery. *arXiv preprint arXiv:2511.02824*, 2025.

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.

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

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

A AlphaEvolve Problems

Table 2 lists the 9 problems studied, taken from Novikov et al. [2025].

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 2: 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].

B 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 IID 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 C. This makes sampling valid solutions feasible but may give these baselines an advantage.[11] 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 1 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.

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.

C 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 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.

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

D 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 1 and 60 seconds for those in Appendix B.

````
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.
````

E Bootstrap Estimate

IID Random Sampling. Here the bootstrap estimate amounts to calculating a pass@k. Under a given budget we find the last generation ShinkaEvolve reached before exceeding it, and as its score take the best bound found until then. If the budget allows sampling k IID programs then the probability of sampling one that matches or outperforms Shinka's score can be approximated by repeatedly sampling k programs out of the n total generated, here being 2000, and seeing if any match or exceed Shinka's score. Here $n = 2000$ and, if c is the number of programs that match/exceed code evolution's score this probability estimate can be analytically calculated as $1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}$ as per Chen et al. [2021]. The average IID cost per program is estimated as the average cost over all sampled programs, so small cost differences between individual generations are ignored.

Sequential Random Search. For the sequential RS bootstrap estimate it is important to consider a realistic scheme of how a given budget would be exhausted, with different setups potentially yielding different results. Here it is assumed that a budget is used up, generation by generation, until a trial is exhausted, after which a new trial is begun. The bootstrap is estimated by picking a random trial, seeing up until which generation in it can be searched under the given budget, and if the trial is exhausted then continuing to one of the remaining trials.

Formally, if $p_{\text{RS}\geq\text{Shinka}}(B, T)$ is the probability of matching/exceeding a baseline score s given a budget B and a set of trials T, where trial $t \in T$'s budget is b_t, then $p_{\text{RS}\geq\text{Shinka}}$ is recursively defined as

$$p_{\text{RS}\geq\text{Shinka}}(B, T) = \frac{1}{|T|} \sum_{t \in T} \max(\mathbb{1}_{\max(t)\geq s}, p_{\text{RS}\geq\text{Shinka}}(B - b_t, T \setminus \{t\})) \tag{1}$$

This assumes that all T trials are within the budget, where in practice some might not be while others are. In these cases the sum is only over the trials within budget. For minimization problems the $\max(t) \geq$ should be changed to $\min(t) \leq$.

F Figure 2 Per-problem Breakdown

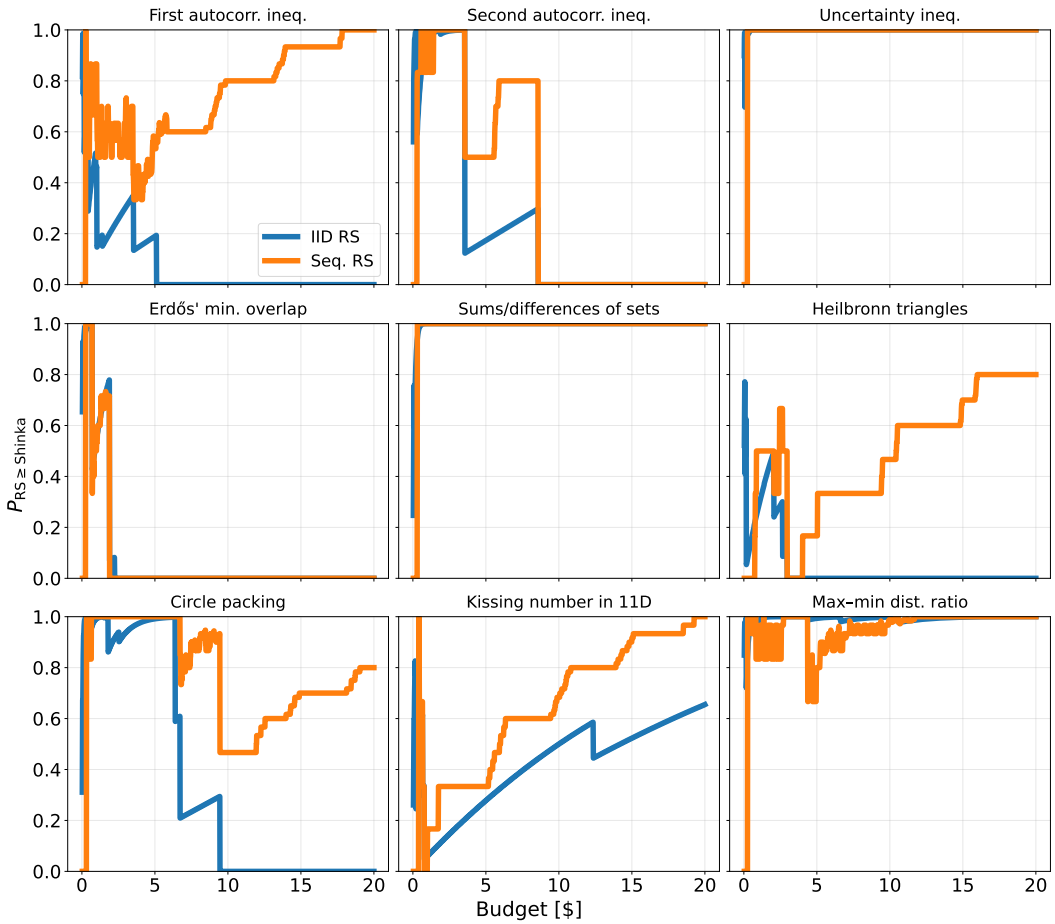


Figure 3: Per-problem probability of matching/exceeding ShinkaEvolve for the two baselines.

G Different Methods' Sample Efficiency

Figures 4 and 5 show respectively the aggregate and per-problem probability of RS matching/exceeding Shinka when the budget is defined not by the API cost but the number of evaluated programs, effectively comparing different methods' sample efficiency. ShinkaEvolve evaluates one program per generation while the RS baselines evaluate one program per sample. Note that for a set API budget Shinka runs until different numbers of generations, as the average cost per program differs per problem.

Shinka being a bit cheaper per program, perhaps due to also using some cheaper LLMs, results in the final probabilities in Figure 4 being lower than those in Figure 2, but only slightly.

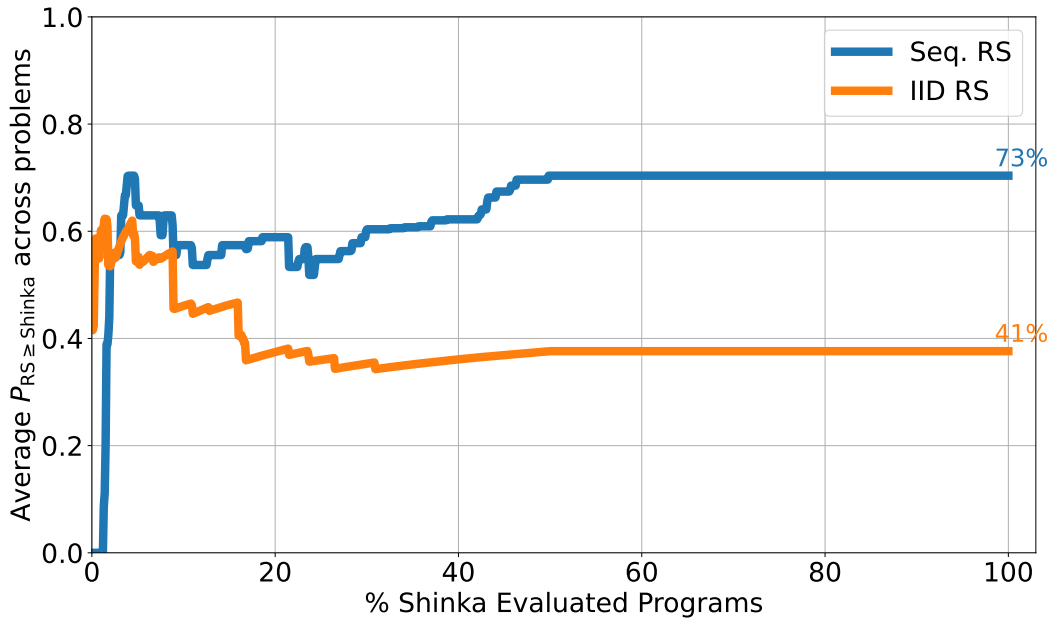


Figure 4: For each baseline, the average probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs. The x axis is the percent of evaluated programs out of the maximum in the corresponding ShinkaEvolve run.

H ShinkaEvolve Hyperparameters

ShinkaEvolve uses the same prompts as the IID RS baseline, 5 island subpopulations due to the slightly larger budget than the default setup, and uses just 2 programs from its archive for inspiration instead of the default 4 in order to reduce API costs.

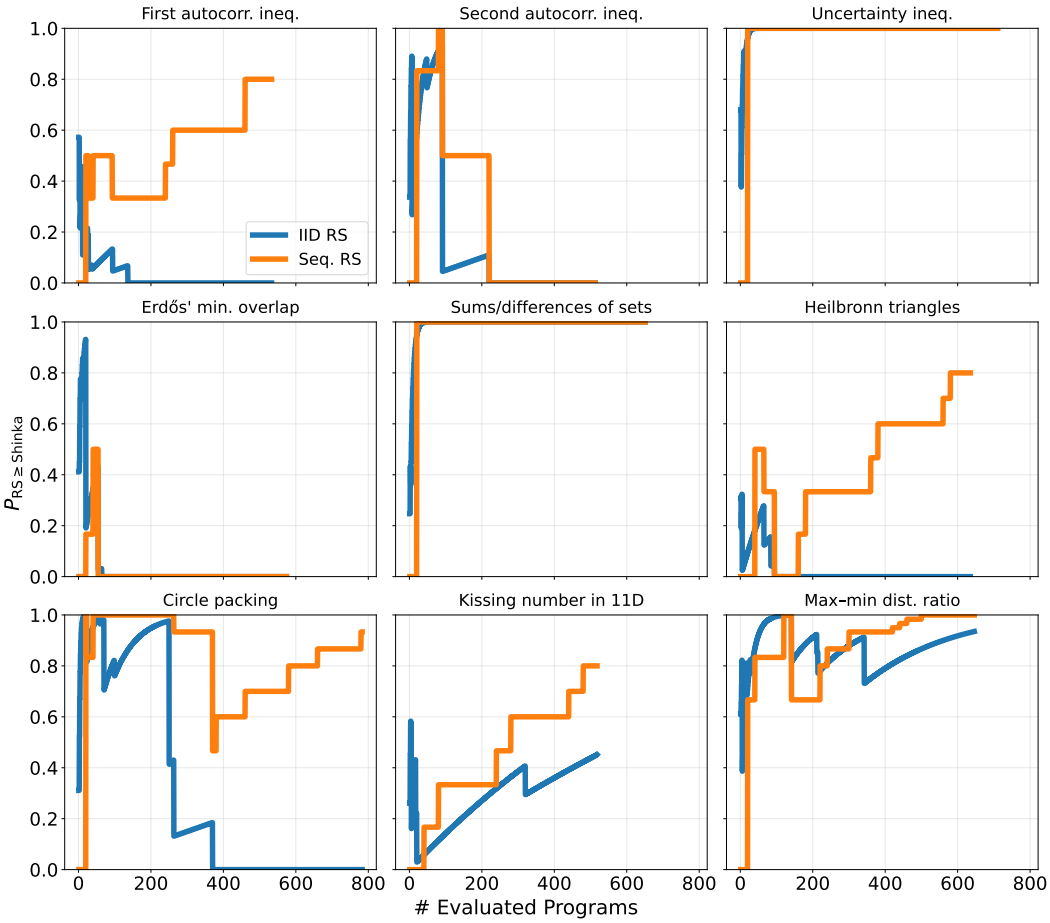


Figure 5: For each baseline, the per-problem probability of matching or exceeding ShinkaEvolve across the 9 problems, as a function of the number of evaluated programs.