Even with AI, Bijection Discovery is Still Hard: The Opportunities and Challenges of OpenEvolve for Novel Bijection Construction

Helen Jenne HELEN.JENNE@PNNL.GOV
Pacific Northwest National Laboratory

Davis Brown
Pacific Northwest National Laboratory
University of Pennsylvania

Jesse He
Pacific Northwest National Laboratory
University of California San Diego

Max Vargas
Pacific Northwest National Laboratory

Henry Kvinge HENRY.KVINGE@PNNL.GOV
Pacific Northwest National Laboratory
University of Washington

Abstract

Evolutionary program synthesis systems such as AlphaEvolve, OpenEvolve, and ShinkaEvolve offer a new approach to AI-assisted mathematical discovery. These systems use large language models (LLMs) to generate candidate solutions to a problem as human readable code, which is then evolved to improve beyond single-shot outputs. While existing mathematical applications have mostly focused on problems of establishing bounds (e.g., sphere packing), the program synthesis approach is well suited to any problem where the solution takes the form of an explicit construction. With this in mind, in this paper we explore the use of OpenEvolve for combinatorial bijection discovery. We describe the results of applying OpenEvolve to three bijection construction problems involving Dyck paths, two of which are known and one of which is open. We find that while systems like OpenEvolve show promise as a valuable tool for combinatorialists, the problem of finding novel, research-level bijections remains a challenging task for current frontier systems, reinforcing the need for human mathematicians in the loop.

Keywords: AI for math, Combinatorial bijections, LLM program synthesis

1. Introduction

Evolutionary systems have recently emerged as powerful tools for scientific discovery, leveraging the ability of large language models (LLMs) to generate and modify code. For example, FunSearch was applied to problems in combinatorial optimization such as the cap set problem (9; 18). Its successor AlphaEvolve discovered faster matrix multiplication algorithms, improvements to the minimum overlap problem, and an improved construction of kissing numbers in dimension 11 (16). These successes were extended in (10) to include advances in finite field Kakeya and Nikodym sets, Crouzeix's conjecture, and the moving

soft problem. These and other evolutionary systems (6; 12; 13) have also been applied to problems in systems research and algorithmic design.

Evolutionary program synthesis approaches use LLMs to mutate and recombine previously generated code. Using code as a medium to develop machine learning-driven solutions facilitates interpretability and may bias towards simpler solutions. Current systems of this kind can be applied to problems satisfying two requirements: (1) the problem can be formulated so that candidate solutions are expressible in a common programming language (e.g., Python), and (2) candidates can be verified automatically.

Compared to other areas of math, problems in algebraic combinatorics are particularly well suited for such systems (5). The objects of interest are often straightforward to represent on a computer, and the software library Sage (23) simplifies their manipulation and analysis. **In this paper, we focus on the problem of *bijection discovery*: given finite sets A and B (often parameterized by a positive integer n), the goal is to produce a mathematically meaningful Python function defining an algorithmic bijection between A and B.** We choose problems where the sets A and B are easily generated in Python to enable automatic verification of the proposed bijections.

There is reason to be optimistic that evolutionary systems could successfully discover bijections. LLM training sets likely contain extensive descriptions of key combinatorial bijections in both natural language (via arXiv and MathOverflow) and code (via Sage), providing prototypes for the kinds of constructions that mathematicians value. Furthermore, LLMs can iterate through potential solutions far more rapidly than humans and draw from a vast breadth of mathematical knowledge. However, whether they are sufficiently creative for this type of discovery remains an open question.

In this paper we present case studies applying the system OpenEvolve (20) to three bijection discovery problems: two with known solutions and one open problem.[1] All problems involve Dyck paths, North East (NE) lattice paths from $(0, 0)$ to (n, n) that stay weakly above the diagonal $y = x$. We chose Dyck paths because their numerous bijections with other objects counted by the Catalan numbers are well-documented in both literature and code, ensuring strong representation in model training data. The problems are:

(1) **Bijection between odd-diagonal avoiding paths and Dyck paths**: NE lattice paths from $(0, 0)$ to $(2n, 2n)$ that avoid the points $(2i - 1, 2i - 1)$, $1 \leq i \leq n$ are enumerated by the Catalan number C_{2n} (22, Problem A4). Stanley (22) rates this problem as difficulty 2+, which he describes as "about the hardest problem that could be reasonably assigned to a class of graduate students". While a bijective proof is known, direct LLM prompting failed to produce the solution, making it an appropriate benchmark for evolutionary methods.

(2) **Bijection between 321-avoiding permutations and Dyck paths**: Although there are at least three known bijections between Dyck paths and 321-avoiding permutations (4), they are nontrivial. We selected this problem based on preliminary experiments where we prompted LLMs to produce bijections between Catalan objects without evolutionary refinement. While even weaker models successfully generated some

1. Code to reproduce our experiments is available at https://github.com/pnnl/OpenEvolve-Bijection Discovery.

bijections (such as between Dyck paths and 213-avoiding permutations), only GPT-5 produced a bijection with 321-avoiding permutations (the Billey–Jockusch–Stanley bijection (3)). The difficulty of this problem suggests it could be a useful case study for OpenEvolve when used with other LLMs.

(3) **Area-bounce exchanging bijection on Dyck paths**: A recent paper (2) defines an area-bounce exchanging bijection on a subset of Dyck paths. This problem is well-suited for OpenEvolve because the existing implementation (24) serves as a starting point, and extending this bijection to a broader class of Dyck paths would be progress on an open research question.

Our experiments yield mixed but instructive results. OpenEvolve successfully discovers the known bijection for problem (1), but surprisingly fails to find a bijection for problem (2) under a variety of configurations (despite GPT-5 finding a bijection when prompted directly). Similarly, for problem (3), the system does not find a solution. These failures provide guidance for future applications and inform this paper's focus on synthesizing lessons for mathematicians considering using evolutionary approaches in combinatorics research. The paper is organized as follows: Section 2 describes OpenEvolve, Section 3 presents our case studies, and Section 4 discusses lessons learned for problem selection, objective function design, and realistic expectations when using these tools in mathematical research.

2. Background

2.1. Evolutionary program synthesis

Evolutionary program synthesis systems like OpenEvolve use LLMs to generate computer programs to optimize a specific objective function. For example, if the objective is sphere packing, each generated program might specify a configuration of spheres which can each be judged by their sphere density. A population of these programs is generated and then evolves as sub-optimal solutions are discarded, solutions are mutated (also via LLMs) to introduce novel features, and new programs are generated. Given the power of modern machine learning, it is reasonable to ask why one should go through the intermediary of code rather than optimizing for a solution directly. Forcing AI systems to produce solutions as computer programs leads to intrinsic interpretability since an expert can directly examine the code, a critical feature when mathematical discovery is the goal. Being able to run a solution as a computer program also allows for verification of correctness (though not at the level of a proof), thus avoiding the common situation where LLMs produce extremely plausible sounding natural language solutions that contain subtle errors. Moreover, requiring solutions to be written in code injects a powerful simplicity bias which avoids 'bag of heuristic' solutions where a model simply enumerates an exhaustive list of conditions that offer no mathematical insight (15).

The basic OpenEvolve algorithm proceeds as follows:

(1) Start with an initial Python program that takes as input an element of a set $A(n)$ for $n \in \mathbb{Z}_{>0}$, and outputs an object in set $B(n)$.

(2) Evaluate this program using an objective function designed to measure how close it is to being a bijection.

(3) Add the program and evaluation to a database ('*population*') of programs.

(4) Prompt an LLM to generate modifications to the program aimed at improving its evaluation.

(5) Apply the LLM's modifications to create a new program and evaluate it.

(6) Repeat steps (4) and (5) for a user-specified number of iterations so as to *evolve* new solutions.

The initial Python program. For each problem, we started with multiple initial Python programs, evolved in parallel using independent populations (called 'islands'). We obtained the initial Python programs by prompting an LLM with a description of the problem, sampling with the default hyperparameters in the chat interface.

Program evaluation. We evaluated programs by (1) running the program on all objects in set $A(n)$ and measuring statistics that quantified how closely it approximated a bijection and (2) prompting an LLM to evaluate the program.

Empirical evaluation for a fixed n. For a small n, we scored each program f by its proximity to being a bijection using three metrics: a *surjectivity score* (the ratio $|f(A(n))|/|B(n)|$), an *injectivity score* (the ratio of unique elements to total elements in $f(A(n))$), and the *validity score* (the fraction of $f(A(n))$ contained in $B(n)$). We averaged the three scores to obtain a single combined score.

LLM evaluation. While we use the word "bijection" in this work and in the broader community, it is understood that what we actually mean is something more specific. Assigning a random pairing between elements in two sets is of limited value. What we really want is a map $A(n) \to B(n)$ that both generalizes to most n and also yields insight into $A(n)$ and $B(n)$. To get at these less quantifiable properties, we used an LLM judge to perform additional checks:

(1) Check for "cheating". We observed that LLMs readily exploit shortcuts to achieve high scores. For example, models sometimes generated all objects from sets $A(n)$ and $B(n)$ and established a one-to-one correspondence using an index-based mapping. The LLM evaluator was instructed to give scores of 0 in these cases.

(2) Evaluate whether the code implemented the algorithm described in the program's docstring[2], in order to differentiate programs that scored low in the empirical evaluation due to a flawed algorithm, and programs that scored low due to implementation errors. The LLM was instructed to give a score on a scale of 0.0 to 1.0 along with an argument that the code matches the description in the docstring, or a counterexample.

(3) To impose a simplicity prior (on a scale of 0.0 to 1.0), the LLM was prompted to give high scores to programs that are simple, intuitive, and reveal a structural correspondence between the two sets, and low scores to programs that seemed convoluted, arbitrary, or involved a lot of case work.

2. When generating programs, LLMs must produce a docstring describing what the program does.

The LLM was also prompted to provide suggestions for improvement. These responses were used in the prompts in the evolution step, in order to give more guidance than a numeric score. The program's final score was a weighted average of the averaged LLM scores and the empirical evaluation.

Prompt for revision. In the evolution step, an LLM is given a prompt that explains the desired bijection (including details around $A(n)$ and $B(n)$), provides hints, describes the evaluation criteria, specifies the program to be edited, and highlights other high-scoring and/or diverse programs in the current database to take inspiration from.

Program sampling OpenEvolve uses the Multi-Dimensional Archive of Phenotypic Elites (MAP-Elites) algorithm (14) to maintain a diverse population of programs, which helps avoid getting stuck in local optima. The algorithm populates a grid defined by program features, scoring only the highest-scoring program in each cell. Programs are selected for evolution by sampling from this grid.

2.2. Related Work

A number of recent works have applied evolutionary program synthesis tools to problems in research-level mathematics. In a follow-on to (16), Georgiev et al. (10) applied AlphaEvolve (the closed-source inspiration for OpenEvolve) to 67 different problems in mathematics with impressive success: in many cases the system was able to discover the best known result and sometimes improved upon it. Notably however, most problems in this work amount to establishing tighter upper or lower bounds on a numerical quantity. Several of their observations are applicable to our use-case and will be discussed further in Section 4.

There are other computational tools to assist mathematicians with bijection discovery, including databases like the OEIS (17) and FindStat (19), and more recently the Bijectionist's Toolkit (11). These tools have been applied to study homomesy (7; 8) and cyclic sieving (1). We see these approaches as complementary and envision a future iteration where LLMs can use these tools in their search for bijections.

3. Case studies

3.1. Odd-diagonal avoiding paths and Dyck paths

Problem. Produce a program implementing a bijective map from NE lattice paths from $(0, 0)$ to $(2n, 2n)$ that avoid the points $(2i - 1, 2i - 1)$ to Dyck paths from $(0, 0)$ to $(2n, 2n)$.

Results. An evolving population of programs generated by OpenEvolve (including a valid solution) is visualized in Figure 1. The top left plot shows the progression of program metrics: the system discovers the bijection after 60 iterations, but the combined scores of new introduced programs are very noisy. The best program validity score within the population is maximized early in evolution, indicating that the LLMs are easily able to generate programs that reliably output Dyck paths. On the other hand, surjectivity and injectivity take many more generations to optimize, with understandable trade-offs occurring between these. The program defining the valid bijection which is generated at iteration 60 can be found in Figure 2 in the Appendix. This is the same as the known bijection from (22). Notably, the docstring was over 600 words and included a sketch of invertibility (see Figure 3

in the Appendix), illustrating how the docstring produced by the system could aid a human mathematician in proving a bijection, provided that it is carefully checked.

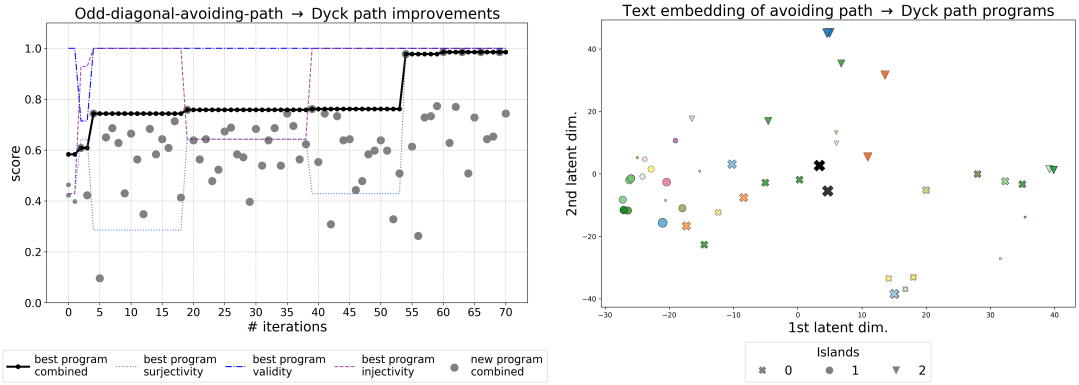


Figure 1: Left: Progression of the best program metrics over 70 iterations for different terms in the scoring function. Right: Text embedding of all programs, after applying dimensionality reduction using PCA. Each functionally distinct mapping is given a different color (the bijection is black). Surjectivity is indicated by the relative size of the points.

How much variation is there in proposed solutions? Surprisingly, we found limited variation in injectivity, surjectivity, and validity scores, suggesting that models tend to gravitate toward a handful of different candidate maps. Further analysis showed that these metrics were not granular enough to always differentiate distinct mappings: the 73 programs produce 26 functionally different mappings but only 10 different surjectivity scores. To further understand the variability among programs with the same metrics and even the same empirical mappings, we plot the text embeddings[3] of all 73 programs after dimensionality reduction (Figure 1, right), giving programs defining the same mapping the same color. We find that programs with the same surjectivity score and even those that functionally correspond to the same map could have very different text embeddings (for example, the two orange upside down triangles that are the exact same program are distant on the plot). This highlights an axis of variation that can be easy to overlook: a significant amount of variation introduced by LLMs can relate to program implementation rather than functionality.

3.2. 321-avoiding permutations and Dyck paths

Problem. Produce a program that implements a bijective map from Dyck paths from $(0, 0)$ to (n, n) to 321-avoiding permutations in S_n.

Results. We applied OpenEvolve with a variety of configurations, but no run yielded a bijection. Figure 4 in the Appendix shows results from two runs, one where the team of

3. We map programs into $\mathbb{R}^{3072}$ using OpenAI's text-embedding-3-large model. Vector proximity reflects textual semantic similarity; programs with similar structure, variable names, and comments will be close.

LLMs included more large models and one where the team included more small models[4]. In the former case, the best program mapped onto the subset of permutations that is both 321-avoiding and 3142-avoiding. For $n = 4$, this program produces 13/14 target permutations, a significant improvement over the initial program which produces only 8/14 target permutations. In contrast, the small model team failed to improve upon the empirical scores of the best initial program. Interestingly, in both runs some responses mention the Billey–Jockush–Stanley bijection (3) by name, but attribute it to the wrong map.

LLMs can struggle when there are several closely related problems. In addition to being in bijection with 321-avoiding permutations, Dyck paths are in bijection with stack-sortable (231-avoiding) permutations, and 132-, 213-, and 312-avoiding permutations. Bijections to 132-, 213-, and 312-avoiding permutations can be obtained from the bijection to stack-sortable permutations by simple operations (reversing the permutation, replacing each value x in the permutation by $n+1-x$, or combining these operations, respectively). *The prominence of these known bijections appears to be a major barrier preventing the models from generating the bijection to 321-avoiding permutations.* The existence of these few prototypical solutions may partly explain the more densely clustered program text embeddings in Figure 4 compared to Figure 1.

3.3. Area-bounce exchanging bijection

Problem. Produce a program that implements a bijective map from Dyck paths to Dyck paths that exchanges the area and bounce statistics.

Set-up details. Since this is an open problem, we seeded the population with the recent map due to Ayyer and Sundaravaradan (2). The system is instructed that the provided map is a valid bijection that exchanges area and bounce on a subset of Dyck paths, and is asked to analyze its failures and propose a targeted modification to generalize it. Programs were scored using paths from $(0, 0)$ to $(4, 4)$. The Ayyer-Sundaravaradan mapping hits 8 of the 14 Dyck paths, is injective, and doesn't produce any out-of-domain outputs, since it returns None if the Dyck path input is not one of the paths the map works for.

LLMs can struggle when constructions in the problem look similar to constructions seen during pretraining. Concepts that seem very salient in LLM's training data interfered with its ability to make progress on this problem. For example, the definition of bounce used in Ayyer and Sundaravaradan (2) is formulated in a way that is slightly non-standard, so the initial feedback from the LLM evaluators was focused on re-defining the bounce statistic. We encountered a similar issue due to the prominence of the zeta map in related literature. The zeta map sends the statistics (area, dinv) to (bounce, area). While this does not solve the problem, the models frequently stated that it did and replaced the entire program with a program that used it. We addressed both of these issues with careful prompting. Ultimately, the model produced a higher scoring solution but not using an algorithmic bijection, which we discuss further in Section 4.

4. Smaller, cheaper models allow for more iterations (and thus more candidate solutions) for the same cost.

4. Lessons learned

Challenge #1: Designing an objective function. A primary challenge in applying evolutionary search for bijection discovery is objective function design. Combinatorial bijections are all-or-nothing constructions: either a map is a bijection or it is not. As such, the community has spent much less time thinking about how to quantify the extent to which an arbitrary map is a bijection. The ability to construct granular scoring functions (ideally continuous) was identified as a crucial ingredient to finding solutions with AlphaEvolve in (10). In our case studies, programs that defined distinct mappings yielded identical surjectivity and injectivity scores, resulting in a plateaued objective landscape.

It may be that the problem of bijection discovery is simply less amenable to incremental improvement than problems like sphere packing, as the discovery of a correct mapping can require a single critical insight that resolves multiple structural flaws at once. Alternatively, program evaluation may simply need refinement with more granular objective functions, such as a score that differentiates between a mapping with many 2-way collisions and one with a single k-way collision.

Challenge #2: Aligning the objective with mathematical intent. Models often discover "loopholes" in the objective function, producing programs that achieve high scores but are not useful from the perspective of mathematical discovery. This is referred to as the "cheating phenomenon" (10) or "reward hacking" (12; 21), and occurs when the solution satisfies the literal constraints but violates the spirit of the problem. As we discussed in Section 2, one cheating behavior that we observed was LLMs exhaustively enumerating all objects from the sets A and B in order to construct a trivial index-based mapping. To prevent this kind of solution, we added explicit instructions not to generate all of the objects in A and B. Additionally, we instructed the LLM evaluator to check for cheating, giving specific examples of what this might look like.

While these steps successfully prevented this particular instance of reward hacking, we found that it was difficult to anticipate all of the different ways a model might reward-hack. For instance, in evolving the area-bounce exchanging bijection (Problem 3), the system implemented a function that, given a Dyck path, first found its area and then used breadth-first search to find a path with the same bounce. While this program resulted in a near-perfect evaluation score, a mathematician would immediately disqualify such a solution, since it is a search algorithm rather than an algorithmic bijection and fails to provide any structural insight.

For the foreseeable future, we believe that in bijection discovery problems an expert-in-the-loop is essential. Their role is not just to define the initial problem and objective, but also to periodically inspect the top-performing solutions for pathological behavior and evaluate whether the system is capturing the intention of the problem.

Challenge #3: The limitations of LLMs as evaluators. The success of these systems currently depends on having objective, machine-verifiable evaluation metrics. While LLMs can be used to evaluate programs, two issues prevent them from being reliable evaluators in this context: the difficulty of capturing mathematical nuances in prompts, and the inconsistency of their responses.

Aside from the case studies described in this paper, we also attempted to apply OpenEvolve to the problem of discovering a *combinatorial interpretation*: given a formula, produce a program that constructs a family of combinatorial objects counted by this formula. While we could automatically verify the correctness of the counts of candidate programs, we were not able to get reliable LLM evaluations of the quality of the interpretations. This was likely because (1) it is difficult to define what it means to be an interesting and useful combinatorial interpretation, making the evaluation prompt difficult to write, and (2) the candidate programs are not easy to evaluate even for a human. The candidate programs often contained long docstrings with elaborate definitions that seemed promising, but were ill-defined or nonsensical once one tried to actually construct an example of the object. This highlights a point that is perhaps obvious but still worth stating: **given that these systems generate hundreds of programs, it is crucial to set up the problem and evaluation in such a way that doesn't require reading LLM responses.**

Beyond the challenge of defining subjective criteria, we also observed that LLM-based scoring was not consistent. While it might be expected that numerical evaluations vary depending on the model, we were surprised to find that even when the model was fixed, repeated evaluations of the exact same program gave scores spanning the entire possible range. This unreliability made it challenging to weight the LLM-based portion of the evaluation. When the LLM evaluation is weighted too low, pathological solutions are not sufficiently penalized, but when it is weighted too high, scores that are spuriously high due to inherent randomness interfering with the search. Given these issues, we think a better use of LLMs in the evaluation pipeline would be as filters (for instance, to automatically remove "cheating" programs) rather than as scorers.

Challenge #4: LLMs favor solutions to well-known problems. Beyond the observation that evolutionary program synthesis works best on problems with a smooth objective landscape, a more subtle challenge arises from biases in the LLM's training data. Our case studies illustrate that models have a tendency to get stuck on famous mathematical results that are associated with key words in the problem, even when they repeatedly prove to be unhelpful. For instance, when tasked with discovering a bijection between Dyck paths and 321-avoiding permutations (Problem 2) the model repeatedly produced programs that involved stack-sorting, which is used in well-known bijections between Dyck paths and other pattern-avoiding permutations. Similarly, in Problem 3, the model was drawn to using the zeta map. These observations suggest that in problem selection and prompt writing, one should consider potential "gravitational pulls" and make efforts to steer models away from unproductive paths.

In summary, we remain optimistic about the application of evolutionary systems for bijection discovery, but successful application will require the right kind of problem with a well-designed objective function along with human expertise and guidance. One interesting additional consequence of working with these systems is that they force a mathematician to very directly confront what they value in a mathematical artifact. Monitoring LLMs that have an endless ability to produce mathematics that aligns with our literal instructions but not with what we want has the potential to sharpen our focus on the value of what we produce in our profession.

Acknowledgments

This work was conducted under the Laboratory Directed Research and Development Program at PNNL, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

References

[1] Ashleigh Adams, Jennifer Elder, Nadia Lafreniere, Erin McNicholas, Jessica Striker, and Amanda Welch. Cyclic sieving on permutations: an analysis of maps and statistics in the FindStat database, 2024. arXiv:2402.16251.

[2] Arvind Ayyer and Naren Sundaravaradan. An area-bounce exchanging bijection on a large subset of Dyck paths. *Annals of Combinatorics*, pages 1–29, 2025.

[3] S.C. Billey, William Jockusch, and Richard P Stanley. Some combinatorial properties of Schubert polynomials. *Journal of Algebraic Combinatorics*, 2(4):345–374, 1993.

[4] David Callan. Bijections from Dyck paths to 321-avoiding permutations revisited, 2007. arXiv:0711.2684.

[5] Herman Chau, Helen Jenne, Davis Brown, Jesse He, Mark Raugas, Sara C. Billey, and Henry Kvinge. Machine learning meets algebraic combinatorics: A suite of datasets capturing research-level conjecturing ability in pure mathematics. In *Proceedings of the 42nd International Conference on Machine Learning*, volume 267, pages 7515–7554, 2025.

[6] Audrey Cheng, Shu Liu, Melissa Pan, Zhifei Li, Bowen Wang, Alex Krentsel, Tian Xia, Mert Cemri, Jongseok Park, Shuo Yang, et al. Barbarians at the gate: How AI is upending systems research, 2025.

[7] William Dowling and Nadia Lafreniere. Homomesy on permutations with toggling actions. *Involve*, 18:829–854, 2025.

[8] Jennifer Elder, Nadia Lafrenière, Erin McNicholas, Jessica Striker, and Amanda Welch. Homomesies on permutations: An analysis of maps and statistics in the findstat database. *Mathematics of Computation*, 93(346):921–976, 2024.

[9] Jordan S Ellenberg, Cristofero S Fraser-Taliente, Thomas R Harvey, Karan Srivastava, and Andrew V Sutherland. Generative modeling for mathematical discovery, 2025. arXiv:2503.11061.

[10] Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale, 2025. arXiv:2511.02864.

[11] Alexander Grosz, Tobias Kietreiber, Stephan Pfannerer, and Martin Rubey. A bijectionist's toolkit. *Séminaire Lotharingien de Combinatoire*, 89, 2023.

[12] Robert Tjarko Lange, Yuki Imajuku, and Edoardo Cetin. ShinkaEvolve: Towards open-ended and sample-efficient program evolution, 2025. arXiv:2509.19349.

[13] Fei Liu et al. LLM4AD: A platform for algorithm design with large language model, 2024. arXiv:2412.17287.

[14] Jean-Baptiste Mouret and Jeff Clune. Illuminating search spaces by mapping elites, 2015. arXiv:1504.04909.

[15] Yaniv Nikankin, Anja Reusch, Aaron Mueller, and Yonatan Belinkov. Arithmetic without algorithms: Language models solve math with a bag of heuristics, 2024. arXiv:2410.21272.

[16] Alexander Novikov, Ngân Vu, Marvin Eisenberger, Emilien Dupont, Po-Sen Huang, Adam Zsolt Wagner, Sergey Shirobokov, Borislav Kozlovskii, Francisco J. R. Ruiz, Abbas Mehrabian, M. Pawan Kumar, Abigail See, Swarat Chaudhuri, George Holland, Alex Davies, Sebastian Nowozin, Pushmeet Kohli, and Matej Balog. AlphaEvolve: A Gemini-powered coding agent for designing advanced algorithms. 2025. URL https://deepmind.google/discover/blog/alphaevolve-a-gemini-powered-coding-agent-for-designing-advanced-algorithms/.

[17] OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences. Published electronically at http://oeis.org.

[18] Bernardino Romera-Paredes et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[19] Martin Rubey, Christian Stump, et al. FindStat - The combinatorial statistics database. URL http://www.FindStat.org.

[20] Asankhaya Sharma. OpenEvolve: an open-source evolutionary coding agent. GitHub repository. URL https://github.com/codelion/openevolve. Accessed: 2025-08-10.

[21] Joar Skalse, Nikolaus Howe, Dmitrii Krasheninnikov, and David Krueger. Defining and characterizing reward gaming. *Advances in Neural Information Processing Systems*, 35:9460–9471, 2022.

[22] Richard P Stanley. *Catalan numbers*. Cambridge University Press, 2015.

[23] W. A. Stein et al. *Sage Mathematics Software (Version 10.6)*. The Sage Development Team. URL http://www.sagemath.org.

[24] Naren Sundar and Arvind Ayyer. qtcatalan-bijection. GitHub repository. URL https://github.com/nanonaren/qtcatalan-bijection.

Appendix A. Mathematical background

A.1. Catalan objects

Catalan numbers The Catalan numbers are the sequence $\{C_n\}_{n\in\mathbb{Z}_{\geq 0}}$ defined by

$$C_n = \frac{1}{n+1}\binom{2n}{n}$$

for $n \geq 0$. The first few Catalan numbers are $C_0 = 1$, $C_1 = 1$, $C_2 = 2$, $C_3 = 5$, $C_4 = 14$, $C_5 = 42$. There are hundreds of combinatorial objects enumerated by the Catalan numbers (22); consequently, this sequence gives rise to a vast number of combinatorial bijections.

Pattern avoiding permutations A permutation π contains a pattern τ if some subsequence of π has the same relative order as τ. Otherwise, π is said to avoid τ. For example, the permutation 13254 avoids the pattern 321, since it has no decreasing subsequence of length three. However, 52431 is not 321-avoiding, since 543, 431, and 531 form the forbidden "321" pattern.

Dyck path statistics The *area* of a Dyck path is the number of full unit cells between the path and the diagonal $y = x$. The *bounce path*, as defined in (2), starts at $(0, 0)$ and travels north until it meets an east step of the Dyck path. It then travels east until it meets the diagonal. This process repeats until the bounce path reaches (n, n). Let the points where the bounce path hits the diagonal be $(b_0, b_0), (b_1, b_1), \ldots, (b_k, b_k)$, where $0 = b_0 < b_1 < \ldots < b_k = n$. The bounce statistic is the sum: $\sum_{i=1}^{k-1}(n - b_i)$. For example, the path $NNENEE$ from $(0, 0)$ to $(3, 3)$ has area 2. Its bounce path hits the diagonal at $(0, 0), (2, 2), (3, 3)$, so its bounce statistic is 1.

A.2. Bijection between 321-avoiding permutations and Dyck paths

We describe the Billey-Jockush-Stanley bijection (3), following the exposition in (4). Given a Dyck path defined as a sequence of N and E steps, the *ascent sequence* $\{a_i\}$ is the sequence that gives the numbers of the consecutive N steps. The *descent sequence* $\{d_i\}$ is similarly defined by giving the numbers of consecutive E steps. Then, define the *ascent code* (resp. *descent code*) to be the sequence of partial sums of the ascent lengths $A_i := \sum_{j=1}^{i} a_j$ (resp. descent lengths $D_i := \sum_{j=1}^{i} d_j$). The last element of the ascent and descent codes are omitted since it is always the length of the path. In a permutation σ, an *excedance location* is a value i such that $\sigma(i) > i$, and $\sigma(i)$ is called the *excedance value*. With these definitions established, the bijection is straightforward to describe: given a Dyck path, let A and D be its ascent and descent code, respectively. Let D be the excedance locations, and let $A+1$ be the excedance values. Fill in the remaining entries of the permutation in increasing order.

Appendix B. Overhead of OpenEvolve setup

Practical barriers to entry include the robustness and usability of the available software, and the overhead of problem formulation. The technical barrier has lowered as codebases like OpenEvolve have become more mature, new options are released, and coding agents have improved. Consequently, the primary overhead will likely be problem selection, designing the objective function, and prompt engineering. The latter might be the least familiar for mathematicians new to using these tools, and we found that generating good prompts required significant iteration at first. In general, our experience was that getting these

systems set up with a new problem requires at least a few hours even when you are familiar with the system and the problem, and longer otherwise.

Cost. Finally, since these systems work best with frontier models requiring paid API calls, cost is an obvious concern. Fortunately, cost-per-token at time of writing means that significant compute is within reach of most mathematicians. For example, one run for Problem 2 used Gemini-2.5-flash, o4-mini, Gemini-2.5-pro, and Claude-3.7-sonnet for evolution and Gemini-2.5-flash for evaluation. A run of 200 iterations cost about $10, with each iteration using two API calls: one for evolution (11,000 tokens on average) and one for evaluation (6,000 tokens on average). Of course, this varies depending on the choice of models, but options like ShinkaEvolve (12) allow one to both set a budget and monitor spending in real time.

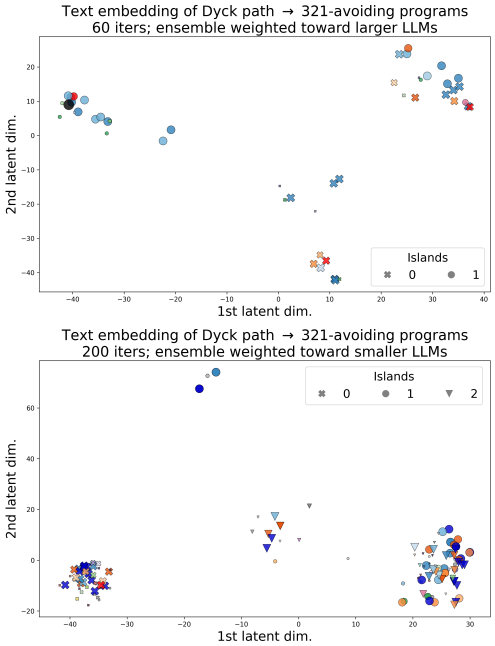
Appendix C. Additional Figures

```python
def avoiding_path_to_dyck_path(path, m):
  # Find the first return to the diagonal: smallest t>0 with height h(t) = 0.
  h = 0
  t = -1
  for i, step in enumerate(path, start=1):
    h += 1 if step == 1 else -1
    if h == 0:
      t = i
      break
  if t == -1:
    raise ValueError("No return to diagonal found; invalid  input.")

  U = path[:t]
  V = path[t:]

  # Determine the Dyck interior X of the primitive excursion U.
  if U[0] == 1:
    X = U[1:-1]  # interior is a Dyck path
    R = avoiding_path_to_dyck_path(V, m-t //2)
    return [1] + X + [0] + R
  else:
    # U = 0 Y 1 with Y a co-Dyck interior
    # take X = complement(Y) which is Dyck.
    Y = U[1:-1]
    X = [1 - s for s in Y]
    R = avoiding_path_to_dyck_path(V, m-t //2)
    return [1] + R + [0] + X
```

Figure 2: The valid bijection between odd-diagonal avoiding paths and Dyck paths produced after 60 iterations.

```
Direct, local bijection by first even-return decomposition with signed interior.

Input/Output:
- Steps are encoded by 1 = North (N), 0 = East (E).
- The input is an odd-diagonal-avoiding (ODA) path from (0,0) to (m,m), i.e., it never visits any
↪  diagonal point (2i-1, 2i-1). Equivalently, if h(t) is the prefix height (#1 - #0). Since the
↪  path avoids all diagonal points (2i-1, 2i-1), any return to the diagonal must occur at an
↪  even-coordinate point (2k, 2k).This means the prefix path to that point has length 4k, so any
↪  return to zero height must occur at a step index t that is a multiple of 4.
- The output is a Dyck path of semilength m: a 0/1-list of length 2m whose prefix heights are never
↪  negative.

Bijection overview:
- Decompose the input path P at its first return to the diagonal (which occurs at an index t  0
↪  (mod 4) because P is ODA):
  P = U · V,  with U a primitive excursion (no diagonal touch inside).
There are exactly two types of primitive excursions U from the diagonal to itself:
  (A) above-diagonal:     U = 1 X 0, with X a Dyck path,
  (B) below-diagonal:     U = 0 Y 1, with Y a "co-Dyck" (i.e., complement of a Dyck path).
In case (B), the complemented interior X := comp(Y) (swap 0 1 in Y) is a Dyck path.
Thus, in both cases we have canonically associated a Dyck path X to the interior of U.

Recursive construction (phi):
- Base: if m = 0 (empty path), return [].
- Find t = first index with height h(t) = 0.
  Let U = path[:t], V = path[t:].
  Let X be the Dyck interior of U defined as:
        if U[0] == 1 then X = U[1:-1]           (above-diagonal excursion)
        if U[0] == 0 then X = comp(U[1:-1])     (below-diagonal excursion)
      Then define
        if U[0] == 1: phi(P) = 1 · X · 0 · phi(V)
        if U[0] == 0: phi(P) = 1 · phi(V) · 0 · X
      This uses only local data at the first even return and recurses on the suffix V,
      which is again ODA (concatenation of ODA excursions).

Why the output is Dyck:
- The word 1 X 0 is a Dyck excursion because X is Dyck. Concatenating Dyck excursions at returns
↪  preserves the Dyck property, so both constructions yield Dyck paths.

Invertibility (sketch):
- Given a Dyck path D, write its unique first-return decomposition D = 1 A 0 B with Dyck A, B. There
↪  are two mutually exclusive cases coming from the forward map:
  (i) D = 1 X 0 phi(V)       (above case), where X = A and phi(V) = B,
  (ii) D = 1 phi(V) 0 X      (below case), where phi(V) = A and X = B.
The inverse identifies which case holds by recursively inverting the left (A) and right (B) Dyck
↪  components and checking which reconstruction produces an ODA primitive excursion first (length
↪  of that excursion must be a multiple of 4 and its interior must be, respectively, Dyck (above)
↪  or complemented Dyck (below)). This test is local and deterministic, giving a two-way inverse
↪  and hence a bijection.

Algorithmic steps implemented here:
1) Validate length and balance.
2) If m == 0: return [].
3) Scan once to find the first index t > 0 with prefix height h(t) = 0 (first even return).
4) Set U = path[:t], V = path[t:].
  - If U starts with 1: X = U[1:-1].
  - If U starts with 0: X = complement(U[1:-1]) (bitwise 1 - s).
5) Recursively compute R = phi(V).
6) Return [1] + X + [0] + R if U[0] == 1, else [1] + R + [0] + X.
```

Figure 3: The docstring of the program in Figure 2.




```python
def map(path):
    n = len(path) // 2
    if n == 0:
        return []

    perm = []
    S = []
    current_num = 1
    for step in path:
        if step == 1:
            S.append(current_num)
            current_num += 1
        elif step == 0:
            last = perm[-1] if perm else -1
            to_move = []
            while S and S[-1] < last:
                to_move.append(S.pop())
            if S:
                perm.append(S.pop())
            perm.extend(reversed(to_move))
    while S:
        perm.append(S.pop())
    return perm
```

Figure 4: Left: Text embeddings of all programs, after dimensionality reduction using PCA (using more large models (**top**) or more smaller models (**bottom**)). Functionally equivalent programs are given the same color. Surjectivity score is indicated by the relative size of the points. Right: The best program produced for the 321-avoiding permutation problem. The docstring and comments have been omitted for space.