EVOLVING TOWARDS BETTER CODES: LLM-GUIDED SEARCH FOR HIGH-DISTANCE BINARY LINEAR CODES

Anonymous authors
Paper under double-blind review

ABSTRACT

Evolutionary program search driven by large language models (LLMs) has produced record-breaking constructions for open problems in combinatorics and beyond. We apply this approach to the longstanding problem of improving the best-known bounds for binary linear codes. Building on the EvoTune evolutionary framework and the ShinkaEvolve codebase, we introduce LINCODEEVOLVE, which evolves code-construction programs against an exact minimum-distance evaluator. A strategy loop combines diversity-driven search and expert supervision: when progress plateaus, new strategies are used to redirect the search. LINCODEEVOLVE discovers seven record-breaking codes, $[172, 21, 66]$, $[173, 20, 68]$, $[176, 21, 68]$, $[181, 21, 70]$, $[184, 21, 72]$, $[189, 22, 72]$ and $[200, 21, 77]$, six of which have concise quasi-cyclic descriptions. With standard code modification techniques, they improve 22 entries of the tables. Every code is verified by exhaustive enumeration; the results have been checked by the maintainer of the code tables and will be incorporated into them. These results suggest that LLM-guided search can help find improved codes and complement existing methods in coding theory.

1 INTRODUCTION

We study the construction of binary linear codes with large minimum distance. This is a central problem in coding theory: minimum distance determines a code's worst-case error-correction capability and enters standard upper bounds on block error probability under maximum-likelihood decoding on the binary symmetric channel (Sason & Shamai, 2006, Eq. (2.4), p. 15). The Gilbert-Varshamov bound establishes the existence of codes with favorable parameters (Varshamov, 1957), yet explicit constructions can exceed this bound for particular lengths and dimensions, as recorded in CodeTables.[1] Finding such constructions improves known lower bounds and can reveal structure shared by codes with large minimum distance.

A construction that works well at one length and dimension need not work equally well at another. Progress therefore calls for selecting, adapting, and combining algebraic constructions and computational search procedures across parameter pairs. This makes code construction a natural setting for automated program discovery: a proposed construction can be executed, its output evaluated, and the result used to guide the next proposal.

FunSearch (Romera-Paredes et al., 2024) implements this idea by pairing language-model-generated programs with an executable evaluator and a database of earlier candidates. High-scoring programs inform subsequent proposals, allowing the search to refine promising ideas while exploring alternatives. FunSearch produced new cap-set constructions and bin-packing heuristics. AlphaEvolve (Novikov et al., 2025) extends program evolution from individual functions to codebases with multiple evolved components. Georgiev et al. (Georgiev et al., 2025) combine this approach with reasoning and proof-assistant systems for mathematical discovery and, in some cases, automated proof. In coding theory, Weindel and Heckel (Weindel & Heckel, 2025) use FunSearch to evolve priority functions for greedy constructions of deletion-correcting codes, recovering the conjectured-optimal Varshamov-Tenengolts construction for a single deletion and improving lower bounds for multiple deletions and quaternary edit codes.

[1] A database of best known lower and upper bounds on the minimum distance of linear codes.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

We introduce LINCODEEVOLVE, a program-evolution framework for discovering binary linear codes with improved parameters. Building on ShinkaEvolve (Lange et al., 2025), LINCODEEVOLVEcoordinates agents that propose and implement experiments with specialist search islands focused on different construction strategies. They draw on a shared bank of strategies and verified codes, initialized with expert-written implementations of diverse coding-theoretic constructions. An executable evaluator assesses proposed codes, and verified results guide later proposals. This separation between proposing a construction and evaluating its output is especially important here: determining minimum distance is computationally difficult in general, while improvements to the best-known bound occur in discrete steps.

Our contributions are a framework for systematically evolving linear-code constructions and new codes that improve longstanding bounds. Each search run targets a fixed length dimension pair (n, k). Within the range $1 \leq k \leq n \leq 200$, LINCODEEVOLVEdiscovers seven parent codes with parameters $[173, 20, 68]$, $[172, 21, 66]$, $[176, 21, 68]$, $[181, 21, 70]$, $[184, 21, 72]$, $[189, 22, 72]$, and $[200, 21, 77]$. Each improves a longstanding lower bound recorded in CodeTables by one to three units. Explicit derivations from these parent codes yield improvements for 22 parameter sets at lengths 171-200 (Section 5.2).

2 BACKGROUND AND RELATED WORK

Binary linear codes Consider $k, n \in \mathbb{N}$, $k \leq n$. A binary linear $[n, k]$ code $\mathcal{C}$ is a k-dimensional subspace of $\mathbb{F}_2^n$. We will refer to $R := \frac{k}{n}$ as the code rate. A full-rank generator matrix $G \in \mathbb{F}_2^{k \times n}$ maps a message vector $u \in \mathbb{F}_2^k$ to a codeword $c = uG \in \mathcal{C}$. By linearity, the minimum distance $d(\mathcal{C})$ is equivalent to the minimum Hamming weight among all nonzero codewords in $\mathcal{C}$:

$$d(\mathcal{C}) := \min_{u \in \mathbb{F}_2^k \setminus \{0\}} \mathrm{wt}(uG) = \min_{u \in \mathbb{F}_2^k \setminus \{0\}} \left| \left\{ i \in [n] \middle| \, (uG)_i = 1 \right\} \right|.$$

The minimum distance establishes the bounded-distance decoding capability of the code, guaranteeing the correction of up to $\lfloor (d(\mathcal{C}) - 1)/2 \rfloor$ arbitrary errors. The multiplicity of weight-w codewords is denoted by A_w, which defines the weight enumerator of $\mathcal{C}$, i.e. the vector $\{A_w | w \in [n]\}$.

Gilbert-Varshamov bound Consider a parameter $\delta \in \left[0, \frac{1}{2}\right)$. Define the binary entropy function $h_2(x) = -x \log_2(x) - (1 - x) \log_2(1 - x)$. Then, for any $\epsilon > 0$ and all sufficiently large n, there exists a binary linear code $\mathcal{C}$ of length n, dimension k, and minimum distance $d(\mathcal{C})$ satisfying

- $k \geq n(1 - h_2(\delta) - \epsilon)$
- $d(\mathcal{C}) \geq n\delta$

At a high level, the Gilbert-Varshamov bound provides an achievability guarantee for the tradeoff between the code rate and relative minimum distance: asymptotically, binary linear codes with relative minimum distance at least δ exist at any rate below $1 - h_2(\delta)$.

Record tables and distance certification. For fixed (n, k), the construction problem is to find a binary linear code with the largest possible minimum distance. Grassl's tables (Grassl, 2006)[2] record the best-known lower and upper bounds on this optimum, together with supporting constructions and references. A construction with certified distance at least d establishes a lower bound; upper bounds limit what any code with those parameters can achieve. These include the Griesmer bound (Griesmer, 1960) and linear-programming bounds.

Certifying a candidate is computationally challenging: computing minimum distance is NP-hard (Vardy, 1997). Exact methods include the Brouwer-Zimmermann algorithm, implemented in Magma and Sagemath (Bosma et al., 1997; Grassl, 2006), and, for small k, enumeration of all $2^k - 1$ nonzero messages. Randomized searches based on information-set decoding (Prange, 1962) can find low-weight codewords and therefore upper bounds on a candidate's minimum distance. However, failing to find a low-weight codeword does not certify its absence. Improving a table's lower bound consequently requires a rigorous distance guarantee.

[2] http://www.codetables.de

Computer search for record codes. Existing searches exploit algebraic structure to make the space of candidate codes more manageable. One major approach searches within quasi-cyclic and quasi-twisted families. This line includes early computational tables (Chen et al., 1969; Gulliver & Bhargava, 1991), structural results for one-generator codes (Aydin et al., 2001), and subsequent search algorithms (Daskalov & Hristov, 2003; Chen, 2015b). A second approach uses *secondary constructions* to obtain new codes from known ones. Examples include Construction X and XX (Sloane et al., 1972), shortening, puncturing, extension, and subcodes (Huffman & Pless, 2003), as well as augmentation methods that add generator rows to a quasi-cyclic parent (Chen, 2015a; Aydin et al., 2017). A third approach prescribes a symmetry group and searches for codes invariant under its action, for example by solving Diophantine systems (Betten et al., 2006) or constructing group codes over semidirect products (Yu et al., 2024). Together, these approaches motivate searching across construction families, parent codes, and symmetries while retaining rigorous distance verification. Learning-based methods have also been explored for specialized families, including reinforcement learning for linear complementary-dual codes (Wu et al., 2025). Our focus is the search for improvements to the general binary linear-code tables.

LLM-driven program evolution. Evolution through Large Models (Lehman et al., 2022) and Language Model Crossover (Meyerson et al., 2023) use an LLM to mutate and recombine programs. FunSearch (Romera-Paredes et al., 2024), AlphaEvolve (Novikov et al., 2025) and ShinkaEvolve (Lange et al., 2025) apply the same loop to open mathematical and algorithmic problems, EvoTune (Surina et al., 2025) guides the mutation through Direct Preference Optimization (DPO) given the scores assigned to the programs by the separately-defined reward function; Wu et al. (2024) survey the field. LINCODEEVOLVE is built on ShinkaEvolve and keeps its loop: sample a parent and a few inspiration programs from an archive split into islands, ask an LLM for a patch or a new program, evaluate it, store it. It also keeps the novelty judge, the meta summarizer, prompt evolution and UCB1-style model selection (Auer et al., 2002). Two things change. The single proposal call becomes a chain of oracle, strategist, contract repair and implementer, with a helper-free invent operator as control. And nothing enters the archive without an exact certificate (Section 4).

Diversity and verified feedback. In our runs the model often handed back the parent matrix unchanged. This matches Gurkan et al. (2026), who show that LLM mutation chains collapse onto a few forms even without selection pressure. Quality-diversity archives (Mouret & Clune, 2015) and island models (Alba & Tomassini, 2002) are the basis of our family-balanced archive and our islands. Learned constructions (Wagner, 2021; Charton et al., 2024) and neuro-symbolic provers (Trinh et al., 2024) share our premise that a learned proposer needs an exact checker. Automated research agents (Lu et al., 2024; Hu et al., 2024) inspired the split of work among our roles. In LINCODEEVOLVE only exact computations reach a score or the stored evidence; what a program says about itself never changes its score.

3 PROBLEM STATEMENT

Fix parameters (n, k) where the table entry is open, meaning the lower bound d_{LB} is strictly less than the upper bound d_{UB}. The task is to find the generator matrix G which generates the binary linear code $\mathcal{C}$ with $d(\mathcal{C}) \geq d_{\mathrm{LB}} + 1$, meaning we beat the best-known linear code (BKLC) for these parameters.

To do this, we track two values: the minimum distance d, and the number of minimum-weight codewords A_d. The latter is called the *shell*. Between two codes with the same minimum distance, we prefer the one with the smaller shell. To reach $d + 1$, the amount of weight-$\leq d$ codewords must be 0, i.e. the shell must be equal to 0, making A_d a useful metric. Minimizing the shell given that the minimum distance is d guides the search in the direction of better codes.

4 THE LINCODEEVOLVE FRAMEWORK

LINCODEEVOLVE evolves construction programs to improve best-known lower bounds on the minimum distance of binary linear codes. It follows the evolutionary search loop of EvoTune (Surina et al., 2025): a language model proposes programs, their outputs are evaluated, and archived results guide subsequent proposals. ShinkaEvolve (Lange et al., 2025) provides the execution infrastructure

and the program database, including islands and migration, novelty assessment and prompt evolution. We retain these components. The code-construction components summarised in Table A are built on top of them, and the language model's parameters remain fixed throughout the search.

Three properties of code construction motivate these additions. First, minimum distance is integer-valued and gives coarse feedback: many candidates share the same distance, so we also compare their numbers of minimum-weight codewords. Second, returning the parent code produces a valid output without advancing the search, so the loop must recognise copies. Third, progress can require a change of construction family, so algebraic tools and evidence about routes already explored must be available when the next proposal is written. At the dimensions we consider, an exact evaluation of one code takes seconds while a model proposal takes minutes, so each generated program should search many candidates during one execution. LINCODEEVOLVE organises this search around five components:

- **An agent team.** The *discover* operator separates proposal generation into three roles: an oracle analyses the current code, a strategist compares construction approaches and commits to one in a structured plan, and an implementer writes the corresponding program. A separate *invent* operator generates programs without direct access to the Toolkit.
- **A Toolkit.** Generated programs can call hand-written strategies that implement constructions and search procedures from the coding-theory literature, so an expert construction costs a few lines rather than a new implementation, and proposals can select, combine and extend them.
- **Evaluation and search feedback.** The evaluator computes the full weight distribution of each admissible returned code. The fitness is the lexicographic order on $(d, -A_d)$: higher minimum distance first, then fewer minimum-weight codewords; it is used for champion selection and parent ranking alike.
- **Specialised islands and construction-based initialisation.** Each island's prompts carry one construction family, such as algebraic codes over extension fields or sparse-graph parity-check design, to encourage exploration of different construction families. The initial code is selected from a portfolio of constructions after exact evaluation of the shortlisted candidates.
- **Exact verification and human input.** Only evaluator-computed results determine a code's fitness; a program's self-reported claims are stored but cannot change it. Human notes, tools and certificates enter through shared files that the prompts read, and interventions are made through explicit pause-and-resume controls.

The model writes the search program: it chooses a construction approach, decides whether to modify the parent or start again, and specifies how candidates are selected. The framework chooses parents and proposal operators, runs the programs, evaluates their outputs, and updates the archive, retaining one program per distinct row space. Figure 1 shows one generation; the following subsections describe its components.

4.1 AGENT-GUIDED PROGRAM GENERATION

Each proposal receives the parent program and generator matrix, its verified weight distribution, the target parameters and distances, applicable bounds, and the island's construction family. Previous results, failed attempts, and mathematical observations such as obstruction bounds and expert notes provide additional context.

Discover. Three successive model calls turn this context into a search program (Figure 2). The *oracle* diagnoses the current code and suggests a move, supported by evidence and a condition for stopping an unsuccessful attempt. The *strategist* compares construction approaches, checks the parameter arithmetic of proposed parent codes, and selects one; it records the approach, the move, whether to modify the parent or start again, the acceptance rule, and the stopping condition in one structured plan. The *implementer* translates this plan into a program that can call the strategy Toolkit and available exact tools. The program must restate the plan and follow its candidate-selection rule.

Invent. A separate operator proposes a procedure without seeing the strategy catalogue or calling its strategies. It first describes the representation, the move, the verification step, and the stopping

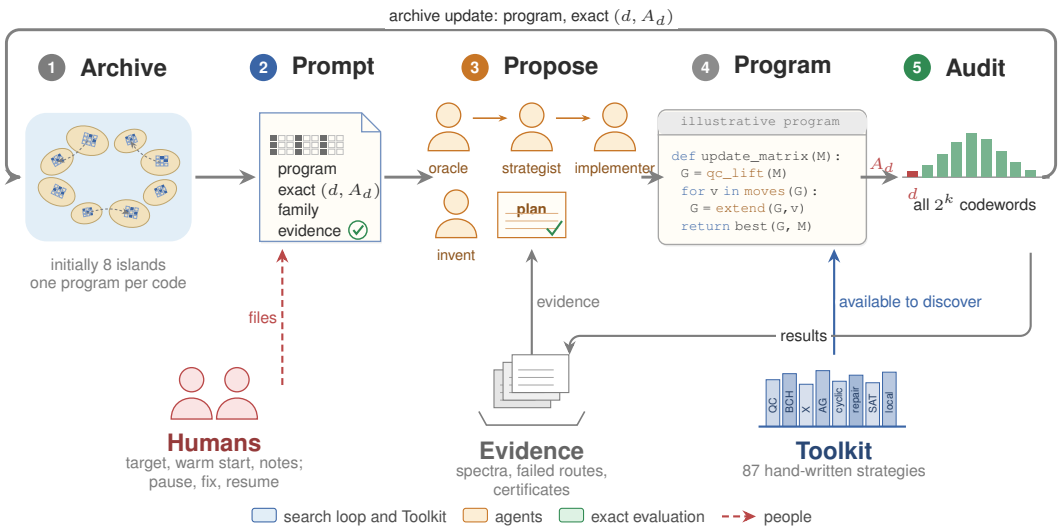


Figure 1: **Agent-guided proposals in LINCODEEVOLVE.** A parent program and its code are selected from the archive (1). The proposal context holds the code's exact (d, A_d), the island's construction approach, and the results of earlier attempts (2). Discover uses three successive model calls to produce a search program; Invent provides an alternative without direct access to the strategy Toolkit (3). The generated program searches candidate matrices, using Toolkit routines when permitted (4). The evaluator checks the returned matrix and computes the exact weight distribution of valid outputs (5). Evaluation results update the archive and inform later proposals. Researchers supply starting points and mathematical observations through shared files.

condition, then implements the procedure from lower-level operations. It still receives the parent program and accumulated evidence, which may reflect earlier use of the strategy Toolkit.

Operator selection. Outside the invention island, proposals are sampled from the inherited *diff*, *full*, and *cross* operators, alongside *discover* and *invent*. Their probabilities adapt to the parent's distance and recent outcomes, with additional weight given to *invent* when the parent is below the distance floor set for the campaign and recent proposals repeatedly return the same code.

Checks and execution. Each model call must return its required fields. If the strategist's plan is incomplete, it gets one more call to complete it, and the proposal is dropped if it is still incomplete. These checks concern the format of the plan, not whether it is mathematically sound. Before a program runs, it is rejected if it only wraps one Toolkit strategy, if it lacks the required header, or if it re-implements the exhaustive enumeration; the novelty judge from ShinkaEvolve, re-prompted for this task, also rejects programs that hardly differ from the parent. A program that passes runs once on the parent's matrix, in its own process, under a time limit. Its output counts only if it is a binary $k \times n$ matrix of rank k; the evaluator then enumerates the code and records (d, A_d). If the program crashes, times out, or returns anything else, it is scored as if it had returned the parent unchanged: it keeps the parent's score, does not enter the archive, and is no longer used as a parent once any program has reached the distance floor. The archive holds up to 96 programs, one per distinct code. A new code is added while there is room; when the archive is full it replaces an archived program under a rule that keeps small construction families represented . A code already in the archive is not added again.

4.2 FITNESS

The fitness of a program is the pair $(d, -A_d)$ of the code it returns, maximized in lexicographic order: a larger minimum distance d is preferred, and at equal distance, fewer minimum-weight codewords A_d win. Both fitness values are exact, from an enumeration of the weight distribution using rbouwer zimmeand alh.

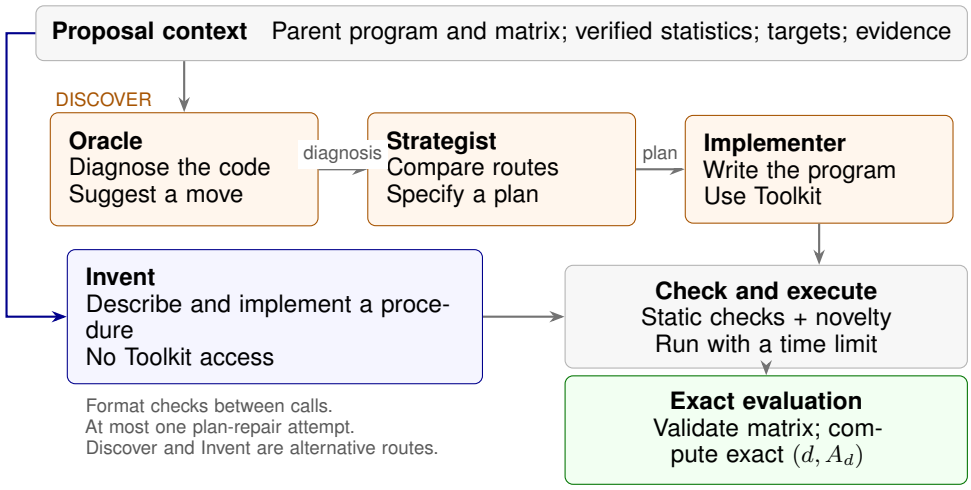


Figure 2: The Discover and Invent proposal operators. Discover separates proposal generation into diagnosis, planning, and implementation. Invent develops a procedure without direct access to Toolkit. Both operators are subject to proposal checks and a time-limited execution. Valid returned matrices are evaluated exactly; failure handling is described in the text.

We use a strictly smaller A_d not only as a heuristic to guide further search but also because it yields a provably more reliable code. Specifically, the undetected-error probability on the binary symmetric channel with crossover probability p is

$$P_{\text{err}}(C; p) \;=\; \sum_{i=1}^{n} A_i \, p^i (1-p)^{n-i},$$

which represents the probability that the channel turns a transmitted codeword into another codeword. For small values of p, this probability is heavily dominated by the leading term $A_d p^d (1-p)^{n-d}$, meaning minimizing A_d directly minimizes the likelihood of undetected errors.

4.3 Initialization: Generation 0

Our search is initialized with a strong initial program that outputs the linear code with the minimum distance exceeding the Gilbert-Varshamov (GV) bound. To construct this base, we evaluated a diverse portfolio of structured code families, including design-based, quasi-cyclic, and algebraic constructions. Candidate codes are first ranked using computationally inexpensive distance estimates [citation needed], and the most promising candidates undergo an exact evaluation using the Brouwer-Zimmermann algorithm implemented in SageMath. We select the code with the largest minimum distance, using the number of low-weight codewords as a secondary filter.

Table 1 compares the selected seeds against the GV bound and random full-rank matrices, reporting both the best and the median minimum distances across 50 random draws. For instances with $k \leq 30$, exact distances are rigorously verified via full enumeration. This initialization establishes a highly competitive baseline from which the subsequent search phase seeks further improvements.

4.4 Specialised Islands

We use ShinkaEvolve's island model to explore several approaches to code construction in parallel. The search begins with eight populations of programs. Each island receives a short prompt describing a construction approach, such as algebraic codes over extension fields, sparse-graph parity-check design, or quasi-cyclic polynomial matrices. The agents use this perspective when proposing changes to the parent code; strategies of other families remain available to them as fallbacks and for final repair.

Two islands have a different role. The cold-start island constructs codes directly from the target parameters (n, k), using the current best matrix only as a fallback. The invention island always uses

the *invent* operator, which has no direct access to the Toolkit. These islands allow the search to explore alternatives to both the current construction and the available Toolkit routines.

Programs migrate between islands every six generations, while the best program on each island is retained. If the best score has not improved for six generations, a new island is seeded from an archived program. This arrangement allows promising programs to spread while maintaining separate searches guided by different construction approaches.

4.5 THE TOOLKIT

The is where most of the human knowledge enters. It is a library of 87 hand-written functions, each taking the current generator matrix and returning a candidate, so a generated program can invoke any of them in one line and combine them freely. They range from general search heuristics, such as simulated annealing and genetic population search, to algebraic constructions from the coding-theory literature: cyclic and BCH parent codes, Construction X, quasi-cyclic and quasi-twisted matrices, and procedures that repair the minimum-weight codewords of an existing code. Six family cores group these strategies by the kind of construction they belong to; a core selects suitable strategies for its input, runs them under a deadline, and returns the best finalist.

Generated programs can combine these strategies in a search loop, choose when to apply them, and decide which candidates to keep. They can also implement additional operations directly. Programs that call only a single strategy without adding search logic are rejected before execution.

cell	LB	GV	random best	random median d	gen 0	lane	s
$[41, 12]$	14	11	$(11, 5)$	10	$(\mathbf{12}, \mathbf{11})$	construction_core	214
$[48, 16]$	16	11	$(11, 3)$	10	$(\mathbf{14}, \mathbf{256})$	qc_core	321
$[50, 20]$	13	10	$(10, 6)$	9	$(\mathbf{11}, \mathbf{30})$	qc_core	413
$[60, 20]$	17	13	$(13, 1)$	12	$(\mathbf{16}, \mathbf{245})$	qc_core	307

Table 1: Generation-0 against random full-rank matrices, exact (d, A_d). Random: best of 50 draws, and the median d of the 50. GV: the Gilbert–Varshamov guarantee for (n, k). the portfolio's pick and its winning lane.

5 RESULTS

5.1 CHOOSING TARGET PARAMETERS(n, k): HUMAN INTUITION

The framework conducts the search at fixed parameters (n, k). We chose these parameters outside the evolutionary loop by examining the code tables, the constructions behind their lower bounds, and the cost of computing the minimum distance. We considered open entries, where there is a gap between the lower and the upper bound . A large gap leaves room for improvement, but may reflect a loose upper bound rather than a weak construction. We therefore used it only as an initial filter.

Plateaus and inherited bounds. We looked for lengths over which the recorded distance stays constant as n increases. At $k = 21$, the lower bound is 64 throughout $165 \leq n \leq 173$ and 68 throughout $179 \leq n \leq 183$ Such plateaus in the minimum distance can arise when a known code is padded with zero coordinates, leaving open the possibility of using the additional coordinates more effectively.

Our number theoretic hint. Our human input was a hint about where to point the search. The row $k = 21$ is frozen at $d = 64$ for $165 \leq n \leq 173$, around $n = 170 = 2 \cdot 85$. For block length 85, $x^{85} - 1$ splits into small factors ($x + 1$, Φ_5 and ten octics), so $k = 21 = 1 + 4 + 8 + 8$ is a natural dimension for an index-2 quasi-cyclic code of length 170. We therefore suggested a quasi-cyclic core of length 170 as a starting point. Similarly, $189 = 3 \cdot 63$ with $22 = 1 + 3 + 6 + 6 + 6$ suggested a starting point for $[189, 22]$.

5.2 New Records

LinCodeEvolve finds seven binary linear codes that improve the lower bounds recorded in the CodeTables snapshot of 25 September 2026. Table 2 summarizes the constructions and their gains. The first five arise from index-2 quasi-cyclic parent codes of length 170 and block length 85; $[189, 22, 72]$ is an index-3 quasi-cyclic code with block length 63. The $[200, 21, 77]$ code is obtained by puncturing a listed $[240, 21, 104]$ binary Goppa-code construction.

Table 2: Seven constructions improving CodeTables lower bounds in the 25 September 2026 snapshot. The gain is the difference between the new minimum distance and the previous lower bound.

Code $[n, k, d]$	Previous bound	Gain
$[172, 21, 66]$	64	$+2$
$[173, 20, 68]$	67	$+1$
$[176, 21, 68]$	66	$+2$
$[181, 21, 70]$	68	$+2$
$[184, 21, 72]$	69	$+3$
$[189, 22, 72]$	71	$+1$
$[200, 21, 77]$	76	$+1$

The constructions have effects beyond their own parameter pairs. Explicit derivations yield improvements in 22 CodeTables entries overall, with gains of up to three in minimum distance.

Figure 3 shows the cluster at lengths 171-191; the $[200, 21, 77]$ construction lies outside its displayed range.

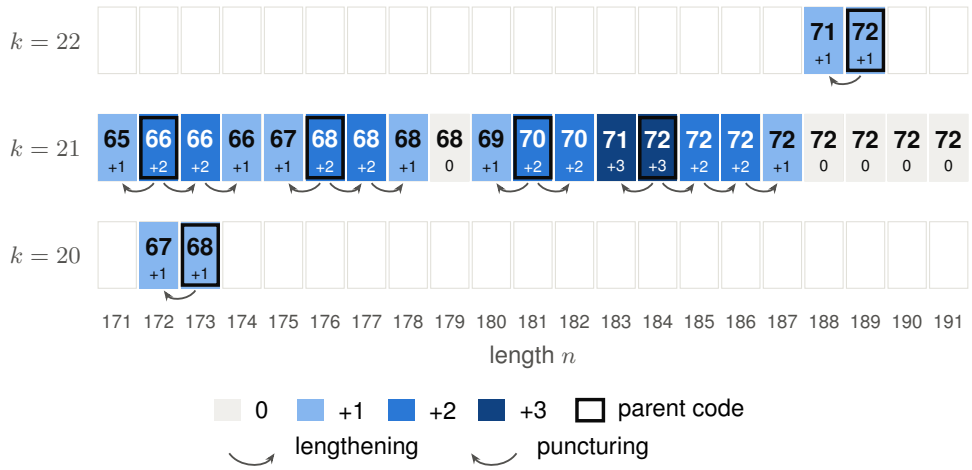


Figure 3: Codes at lengths 171-191. Cells show minimum distance and gain over CodeTables; outlines identify parent codes.

We verified the reported minimum distances using SageMath (The Sage Developers, 2024) and Magma (Bosma et al., 1997). The appendix and accompanying files provide the generator matrices, weight distributions, derivations, and verification scripts.

Markus Grassl, who maintains CodeTables , independently checked the new constructions and accepted them for inclusion. The live tables have not yet been updated. The improvements are measured against the bounds listed in CodeTables on 24 September 2026.

5.3 Structural Advantages of Our Codes

A simpler construction matching the best-known lower bound. LinCodeEvolve also finds codes that match existing distance bounds while admitting concise algebraic descriptions. For example, it constructs a binary $[50, 25, 10]$ code, meeting the current best-known lower bound of

10 (the upper bound is 12). The code is pure double circulant: its generator matrix has the form $[\, I_{25} \mid A \,]$, where A is a 25×25 circulant matrix. This example shows that the search can recover structured, readily reproducible constructions even when it does not improve the distance bound.

Comparison with direct prompting of GPT-6 Astra. For $(n, k) = (172, 21)$, LINCODEEVOLVE finds a code with minimum distance 66 built from a quasi-cyclic $[170, 21, 64]$ core by adding two coordinates via Construction X. The best code obtained by directly prompting GPT-6 Astra at extra-high reasoning effort in our comparison has minimum distance 65.

At $(n, k) = (200, 21)$, both approaches reach distance 77, but they produce inequivalent codes with different weight distributions. The LINCODEEVOLVE code has 25 minimum-weight codewords, compared with 75 for the directly prompted code. It also has fewer nonzero codewords of weight at most 78 (264 versus 303).

Consequently, its parity extension yields a $[201, 21, 78]$ code with 264 minimum-weight codewords, compared with 303 for the extension of the other code.

Direct prompting might recover these constructions with more attempts. The contribution of LINCODEEVOLVE is a systematic search process that can be applied across pairs of parameters without creating a new prompt for each target. It preserves verified candidates, uses exact evaluations to guide subsequent proposals, and produces codes whose properties can be independently checked.

6 CONCLUSION

Using LINCODEEVOLVE, we found seven parent codes using this method. The code modification techniques extend these results to 22 parameter pairs. Although we applied LINCODEEVOLVE only to binary codes, the same approach can be used for q-ary codes, given an exact minimum-distance evaluator over $\mathbb{F}_q$. Since exact evaluation then scales like q^k rather than 2^k, the feasible dimensions are correspondingly smaller. The search can also be restricted to structured families to look for the best codes within them.

The time complexity of our method scales exponentially in terms of code dimension. However, for fixed k, longer codes remain inexpensive to evaluate, so LINCODEEVOLVE may extend to larger lengths. A practical obstacle is comparison: the binary tables of CodeTables currently stop at length 256, which makes it harder to tell whether a longer code beats the best known code of the same length and dimension.

7 LLM USAGE STATEMENT

We used LLM-based chatbots to improve and polish the writing of this paper. Given the nature of our research, we also used LLMs for ideation within the framework, for example, to generate improved prompts based on successful prior results. However, beyond the framework itself, we did not use LLMs to generate ideas or strategies for applying the framework. We also used LLMs to identify potentially relevant citations. All such citations were independently verified by the authors before being included in the literature review. Finally, LLMs were used to assist in drafting portions of the manuscript; these sections were subsequently reviewed, revised, and, where necessary, rewritten by the authors.

REFERENCES

Enrique Alba and Marco Tomassini. Parallelism and evolutionary algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5):443–462, 2002. doi: 10.1109/TEVC.2002.800880.

Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2–3):235–256, 2002. doi: 10.1023/A:1013689704352.

Nuh Aydin, Irfan Siap, and Dijen K. Ray-Chaudhuri. The structure of 1-generator quasi-twisted codes and new linear codes. *Designs, Codes and Cryptography*, 24(3):313–326, 2001. doi: 10.1023/A:1011283523000.

Nuh Aydin, Nicholas Connolly, and John Murphree. New binary linear codes from quasi-cyclic codes and an augmentation algorithm. *Applicable Algebra in Engineering, Communication and Computing*, 28:339–350, 2017. doi: 10.1007/s00200-017-0327-x.

Anton Betten, Michael Braun, Harald Fripertinger, Adalbert Kerber, Axel Kohnert, and Alfred Wassermann. *Error-Correcting Linear Codes: Classification by Isometry and Applications*, volume 18 of *Algorithms and Computation in Mathematics*. Springer, 2006. doi: 10.1007/3-540-31703-1.

Wieb Bosma, John Cannon, and Catherine Playoust. The Magma algebra system I: The user language. *Journal of Symbolic Computation*, 24(3–4):235–265, 1997. doi: 10.1006/jsco.1996.0125.

François Charton, Jordan S. Ellenberg, Adam Zsolt Wagner, and Geordie Williamson. PatternBoost: Constructions in mathematics with a little help from AI. *arXiv preprint*, 2024. arXiv:2411.00566.

Chin-Long Chen, W. Wesley Peterson, and Edward J. Weldon. Some results on quasi-cyclic codes. *Information and Control*, 15(5):407–423, 1969. doi: 10.1016/S0019-9958(69)90497-5.

Eric Zhi Chen. New binary h-generator quasi-cyclic codes by augmentation and new minimum distance bounds. *Designs, Codes and Cryptography*, 2015a. doi: 10.1007/s10623-015-0059-5.

Eric Zhi Chen. A new iterative computer search algorithm for good quasi-twisted codes. *Designs, Codes and Cryptography*, 76(2):307–323, 2015b. doi: 10.1007/s10623-014-9938-4.

Rumen Daskalov and Plamen Hristov. New binary one-generator quasi-cyclic codes. *IEEE Transactions on Information Theory*, 49(11):3001–3005, 2003. doi: 10.1109/TIT.2003.819337.

Bogdan Georgiev, Javier Gómez-Serrano, Terence Tao, and Adam Zsolt Wagner. Mathematical exploration and discovery at scale, 2025.

Markus Grassl. Searching for linear codes with large minimum distance. In Wieb Bosma and John Cannon (eds.), *Discovering Mathematics with Magma*, pp. 287–313. Springer, 2006. doi: 10.1007/978-3-540-37634-7_13.

James H. Griesmer. A bound for error-correcting codes. *IBM Journal of Research and Development*, 4(5):532–542, 1960. doi: 10.1147/rd.45.0532.

T. Aaron Gulliver and Vijay K. Bhargava. Some best rate $1/p$ and rate $(p-1)/p$ systematic quasi-cyclic codes. *IEEE Transactions on Information Theory*, 37(3):552–555, 1991. doi: 10.1109/18.79911.

Can Gurkan, Forrest Stonedahl, and Uri Wilensky. Mutation without variation: Convergence dynamics in LLM-driven program evolution. *arXiv preprint*, 2026. arXiv:2606.05408.

Shengran Hu, Cong Lu, and Jeff Clune. Automated design of agentic systems. *arXiv preprint*, 2024. arXiv:2408.08435.

W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. doi: 10.1017/CBO9780511807077.

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

Joel Lehman, Jonathan Gordon, Shawn Jain, Kamal Ndousse, Cathy Yeh, and Kenneth O. Stanley. Evolution through large models. *arXiv preprint*, 2022. arXiv:2206.08896.

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*, 2024. arXiv:2408.06292.

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

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

Alexander Novikov et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery. *arXiv preprint*, 2025. arXiv:2506.13131.

Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Transactions on Information Theory*, 8(5):5–9, 1962. doi: 10.1109/TIT.1962.1057777.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M. Pawan Kumar, Emilien Dupont, Francisco J. R. Ruiz, Jordan S. Ellenberg, Pengming Wang, Omar Fawzi, Pushmeet Kohli, and Alhussein Fawzi. Mathematical discoveries from program search with large language models. *Nature*, 625:468–475, 2024.

Igal Sason and Shlomo Shamai. Performance analysis of linear codes under maximum-likelihood decoding: A tutorial. *Foundations and Trends in Communications and Information Theory*, 3(1–2): 1–225, 2006. doi: 10.1561/0100000009.

Neil J. A. Sloane, Sudhakar M. Reddy, and Chin-Long Chen. New binary codes. *IEEE Transactions on Information Theory*, 18(4):503–510, 1972. doi: 10.1109/TIT.1972.1054833.

Anja Surina, Amin Mansouri, Lars Quaedvlieg, Amal Seddas, Maryna Viazovska, Emmanuel Abbe, and Caglar Gulcehre. Algorithm discovery with llms: Evolutionary search meets reinforcement learning, 2025.

The Sage Developers. *SageMath, the Sage Mathematics Software System*, 2024. `https://www.sagemath.org`.

Trieu H. Trinh, Yuhuai Wu, Quoc V. Le, He He, and Thang Luong. Solving olympiad geometry without human demonstrations. *Nature*, 625:476–482, 2024. doi: 10.1038/s41586-023-06747-5.

Alexander Vardy. The intractability of computing the minimum distance of a code. *IEEE Transactions on Information Theory*, 43(6):1757–1766, 1997. doi: 10.1109/18.641542.

Rom R. Varshamov. Estimate of the number of signals in error correcting codes. *Doklady Akademii Nauk SSSR*, 117:739–741, 1957.

Adam Zsolt Wagner. Constructions in combinatorics via neural networks. *arXiv preprint*, 2021. arXiv:2104.14516.

Franziska Weindel and Reinhard Heckel. LLM-guided search for deletion-correcting codes. *arXiv preprint arXiv:2504.00613*, 2025.

Xingyu Wu, Sheng-hao Wu, Jibin Wu, Liang Feng, and Kay Chen Tan. Evolutionary computation in the era of large language model: Survey and roadmap. *arXiv preprint*, 2024. arXiv:2401.10034.

Yansheng Wu, Jin Ma, and Shangdong Yang. Linear complementary dual codes constructed from reinforcement learning. *Journal of Systems Science and Complexity*, 38:1388–1403, 2025. doi: 10.1007/s11424-025-4313-2.

Cong Yu, Shixin Zhu, Hao Chen, Yang Li, and Xiuyu Zhang. New record-breaking binary linear codes constructed from group codes. *arXiv preprint*, 2024. arXiv:2412.15551.

A EXPLICIT CONSTRUCTIONS

All polynomials are over $\mathbb{F}_2$. Given a block length m, a divisor h of $x^m - 1$ of degree k, $g = (x^m - 1)/h$ and polynomials $f_1 = 1, f_2, \ldots, f_\ell$, the words

$$\big(bgf_1,\ bgf_2,\ \ldots,\ bgf_\ell\big) \bmod (x^m - 1), \qquad b = \textstyle\sum_{i<k} b_i x^i,$$

form a quasi-cyclic code of length $m\ell$, index ℓ and dimension k (coordinate j of a block is the coefficient of x^j). Some codes append *tail* coordinates after the blocks. These depend only on

$$r = b \bmod (x^5 - 1) = (r_0, \ldots, r_4), \qquad r_t = \sum_{i \equiv t \pmod 5} b_i;$$

a tail coordinate equals $\langle c, r\rangle = \sum_t c_t r_t$ for a pattern c, printed as the bit string $c_0c_1c_2c_3c_4$. In the generator matrix whose row j is the codeword of $b = x^j$ ($0 \le j < k$), its entry in row j is $c_{j \bmod 5}$. The following seven base codes give all improved entries of Table A.

[172, 21, 66]

- $m = 85$, index 2.
- h is the product of
  - $x + 1$;
  - $x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^6 + x^5 + x^4 + x^3 + x + 1$;
  - $x^8 + x^7 + x^5 + x + 1$.
- $f_1 = 1$ and $f_2 = x^{13} + x^{11} + x^9 + x^6 + x^4 + x^3 + x^2 + x + 1$.
- Tail: $(r_0{+}\ldots{+}r_4,\ r_0{+}\ldots{+}r_4)$, i.e. the value $b(1) = b \bmod (x + 1)$ repeated twice. This is Construction X with the repetition code $[2, 1, 2]$.

[173, 20, 68]

- $m = 85$, index 2.
- h is the product of
  - $x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^5 + x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^7 + x^6 + x^5 + x^4 + x^3 + 1$.
- $f_1 = 1$ and $f_2 = x^{15} + x^{11} + x^6 + x^3 + x^2 + 1$.
- Tail: 3 coordinates $\langle c_i, r\rangle$, where c_1, c_2, c_3 are `01001`, `10100`, `01010`.

[176, 21, 68]

- $m = 85$, index 2.
- h is the product of
  - $x + 1$;
  - $x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^6 + x^5 + x^4 + x^3 + x + 1$;
  - $x^8 + x^7 + x^5 + x + 1$.
- $f_1 = 1$ and $f_2 = x^{13} + x^{11} + x^9 + x^6 + x^4 + x^3 + x^2 + x + 1$.
- Tail: 6 coordinates $\langle c_i, r\rangle$, where $c_1, \ldots, c_6$ are `01111`, `11111`, `11111`, `10010`, `11011`, `00101`.

[181, 21, 70]

- $m = 85$, index 2.
- h is the product of
  - $x + 1$;
  - $x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^5 + x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^7 + x^3 + x + 1$.
- $f_1 = 1$ and $f_2 = x^{13} + x^{10} + x^7 + x^6 + x^4 + x^3 + 1$.
- Tail: 11 coordinates $\langle c_i, r \rangle$, where $c_1, \ldots, c_{11}$ are `11000`, `10100`, `11100`, `01010`, `01110`, `10001`, `11101`, `00011`, `10011`, `00111`, `01111`.

[184, 21, 72]

- $m = 85$, index 2.
- h is the product of
  - $x + 1$;
  - $x^4 + x^3 + x^2 + x + 1$;
  - $x^8 + x^6 + x^5 + x^4 + x^2 + x + 1$;
  - $x^8 + x^7 + x^5 + x^4 + x^3 + x^2 + 1$.
- $f_1 = 1$ and $f_2 = x^{13} + x^9 + x^8 + x^7 + x^6 + x^5 + x^3 + x + 1$.
- Tail: 14 coordinates $\langle c_i, r \rangle$, where $c_1, \ldots, c_{14}$ are `11001`, `01001`, `00101`, `11101`, `11110`, `00111`, `11100`, `01010`, `10010`, `10001`, `11111`, `01110`, `10011`, `00110`.

[189, 22, 72]

- $m = 63$, index 3.
- h is the product of
  - $x + 1$;
  - $x^3 + x^2 + 1$;
  - $x^6 + x^5 + 1$;
  - $x^6 + x^5 + x^3 + x^2 + 1$;
  - $x^6 + x^5 + x^4 + x^2 + 1$.
- $f_1 = 1$, $f_2 = x^{14} + x^{13} + x^{12} + x^9 + x^7 + x^5 + x^3 + x^2 + 1$ and $f_3 = x^{16} + x^{14} + x^{13} + x^{10} + x^7 + x + 1$.
- No tail.

[200, 21, 77]

- Generator matrix.
- $A_{77} = 25$, against $A_{77} = 75$ for the $[200, 21, 77]$ code obtained by direct prompting.

[50, 25, 10] (double circulant)

- $m = 25$, index 2.
- $h = x^{25} - 1$, so $g = 1$.
- $f_1 = 1$ and $f_2 = x^{23} + x^{22} + x^{21} + x^{20} + x^{19} + x^{18} + x^{16} + x^{15} + x^{14} + x^{12} + x^8 + x^7 + x^6 + x^4 + x^2 + x$.
- Equivalently, the generator matrix is $[\, I_{25} \mid A \,]$, where A is the circulant matrix whose first row is the coefficient vector of f_2.
- $d = 10$, the best known minimum distance for $[50, 25]$, and $A_{10} = 125$.

ALL IMPROVED ENTRIES.

n	k	d_{old}	d^{ub}	A_d
172	20	66	67	272
173	20	67	**68**	1156
171	21	64	65	2040
172	21	64	**66**	2720
173	21	64	66	680
174	21	65	66	680
175	21	66	67	1088
176	21	66	**68**	2992
177	21	66	68	2720
178	21	67	68	1054
180	21	68	69	1428
181	21	68	**70**	3502
182	21	68	70	2771
183	21	68	71	2108
184	21	69	**72**	9826
185	21	70	72	8670
186	21	70	72	6188
187	21	71	72	6103
200	21	76	**77**	25
201	21	77	78	264
188	22	70	71	924
189	22	71	**72**	2394

Table 3: Improved entries, with the best previously known minimum distance (*old*), the minimum distance of our code (*new*), and the number A_d of minimum-weight codewords of the released code.