LLM-Generated Search Heuristics Can Solve Open Instances of Combinatorial Design Problems

Christopher D. Rosin
https://constructive.codes
christopher.rosin@gmail.com

Abstract

We assess the application of code generation via LLMs to a specific task in the mathematical field of combinatorial design. This field studies diverse types of combinatorial designs, many of which have lists of open instances for which existence has not yet been determined. While systematic constructions and proofs are preferred approaches to addressing open instances, a complementary approach is to develop computational search heuristics specific to each type of design. We introduce a Constructive Protocol (CPro1) that uses LLMs to generate search heuristics. Starting with only a textual definition and a validity verifier for a particular type of design, CPro1 guides LLMs to select and implement strategies, while providing automated hyperparameter tuning and execution feedback. CPro1 successfully solves long-standing open instances for 7 of 16 combinatorial design problems selected from the 2006 *Handbook of Combinatorial Designs*, as well as open instances for 3 of 4 problems from recent (Feb. 2025) literature. We conclude that LLM-based code generation can contribute to mathematical research by solving open instances of combinatorial design problems using heuristic search.

1 Introduction

Combinatorial designs are systems of finite sets that satisfy specified constraints. The particular finite sets and constraints involved define the *type* of combinatorial design (e.g. Balanced Incomplete Block Design, Packing Array). The *existence problem* (or *combinatorial design problem*) for a particular type of design has a small number of input *parameters* (e.g. size), and asks whether it is possible to construct a design that satisfies the constraints for these parameters. The existence problem is often addressed by systematic mathematical constructions that are proven correct and show existence for large classes of instances, and by proofs of impossibility that show certain instances cannot exist. But these may leave behind parameters for which existence is unknown; these remain as open instances.

As an example, a *Symmetric Weighing Matrix* is an $n \times n$ matrix W with entries in $\{0, 1, -1\}$, satisfying $WW^T = wI$ and $W = W^T$. For $w = 9$, the last two open instances were $n = 19$ and $n = 21$ [4, 6]. This paper's results include the first constructions of these two previously unknown matrices, completing the result that a $w = 9$ Symmetric Weighing Matrix exists for every $n \geq 12$.

One standard approach to small open instances is heuristic computational search ([4] chapter VII.6). It requires adapting various heuristics to the combinatorial design problem, experimentally tuning each to determine whether they can construct the desired instances. In this paper, we develop a Constructive Protocol *CPro1* that uses LLMs to generate heuristic search code for diverse candidate methods. The protocol automates an experimental process to identify and optimize heuristic strategies, and has the potential to accelerate resolution of open instances for combinatorial design problems.

Related Work: *FunSearch* [26] uses code-generating LLMs in an evolutionary algorithm to search for greedy functions. It solved an open problem by constructing a new *Cap Set*, and was used to

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: MATH-AI.

Packing Array (PA): an $N \times k$ array with entries in the set $\{0, 1, \ldots v-1\}$, such that every $N \times 2$ subarray contains every ordered pair of symbols at most once.

(N,k,v): ***(24,7,6)*** ***(18,8,6)*** ***(28,10,8)*** ***(24,11,8)*** ***(32,11,9)*** ***(28,12,9)*** ***(21,14,9)*** ***(19,15,9)***

Symmetric Weighing Matrix (SymmW): an $n \times n$ matrix W with entries in the set $\{0, 1, -1\}$, satisfying $WW^T = wI$ and $W = W^T$.

(n,w): ***(19,9)***† ***(21,9)***† ***(22,16)***

Skew Weighing Matrix (SkewW): an $n \times n$ matrix W with entries in the set $\{0, 1, -1\}$, satisfying $WW^T = wI$ and $W^T = -W$.

(n,w): ***(18,9)***

Bhaskar Rao Design (BRD): a $v \times b$ array with entries in the set $\{-1, 0, 1\}$. Each row contains r nonzero entries and each column contains k nonzero entries. For any pair of distinct rows, the pairwise element products contain -1 and $+1$ each $L/2$ times.

(v,b,r,k,L): ***(15,42,14,5,4)*** ***(15,126,42,5,12)*** ***(16,48,15,5,4)***

Balanced Ternary Design (BTD): an arrangement of V elements into B multisets, or blocks, each of cardinality $K \leq V$, satisfying: 1. Each element appears $R = p1 + 2 * p2$ times, with multiplicity one in $p1$ blocks and multiplicity two in $p2$ blocks. 2. Every pair of distinct elements appears L times.

(V,B;p1,p2,R;K,L): ***(17,17;8,2,12;12,8)*** ***(14,21;6,3,12;8,6)*** ***(12,16;4,4,12;9,8)*** ***(16,16;7,3,13;13,10)*** ***(12,21;4,5,14;8,8)*** ***(16,22;9,1,11;8,5)*** ***(21,21;12,1,14;14,9)***

Equidistant Permutation Array (EPA): m rows, each a permutation of $\{0, 1, \ldots n-1\}$. Each pair of distinct rows must differ in exactly d positions.

(n,d,m): ***(12,8,21)***†

Florentine Rectangle (FR): r rows, each a permutation of $S = \{0, 1, \ldots n-1\}$. For distinct $a, b \in S$ and $m \in \{1, 2, ..., n-1\}$, at most one row has b positioned m steps to the right of a.

(r,n): ***(7,20)*** ***(7,24)*** ***(7,25)*** ***(7,26)*** ***(7,27)***

(a) Open instances from 2006 *Handbook*, that were solved and found to exist by CPro1. († indicates the 3 instances that were initially solved by manually-developed local search during creation of the prototyping set.)

Covering Sequence (CS): A cyclic sequence $x_0, x_1, \ldots x_{L-1}$ of length L over the binary alphabet, such that for any length-n binary word there exists a j such that subsequence $x_j, x_{(j+1) \bmod L}, x_{(j+2) \bmod L}, \ldots x_{(j+n-1) \bmod L}$ is Hamming distance at most R from the word.

(n,R,L): ***(9,1,71)*** ***(10,1,138)*** ***(11,1,224)*** ***(11,2,64)*** ***(12,2,127)*** ***(12,3,36)*** ***(13,2,276)*** ***(13,3,61)*** ***(14,3,122)*** ***(15,3,230)*** ***(16,3,426)*** ***(17,2,3938)*** ***(17,3,795)*** ***(18,1,52390)*** ***(18,2,7605)*** ***(18,3,1481)*** ***(19,1,104498)*** ***(19,2,14797)*** ***(19,3,2734)*** ***(20,1,207000)*** ***(20,2,28901)*** ***(20,3,5102)***

Johnson Clique Cover (JCC): The Johnson Graph $J(N, k)$ has a vertex for each k-element subset of $\{1, 2, \ldots N\}$. Two subsets A, B are connected by an edge if $|A \cap B| = k-1$. A size-C Johnson Clique Cover has C cliques in $J(N, k)$ such that their union includes all vertices in $J(N, k)$.

(N,k,C): ***(13,4,105)*** ***(13,6,248)*** ***(14,4,138)*** ***(14,6,410)*** ***(15,4,177)***

Uniform Nested Steiner Quadruple System (UNSQS): A *Steiner Quadruple System (SQS)* is a set of blocks, each a size-4 subset of $V = \{0, 1, ..., v-1\}$, such that each subset of 3 elements of V is contained in exactly one block. A *Uniform Nested SQS* splits each block into two *ND-pairs* of 2 elements each, such that each distinct ND-pair appears the same number of times. Let p denote the number of distinct ND-pairs that appear among the blocks.

(v,p): ***(14,91)***

(b) Open instances from Feb. 2025 articles [3, 18, 2], that were solved and found to exist by CPro1.

Deletion Code (DC): a set of m binary words of length n. Given word x, $D(x)$ is the length-$(n-s)$ words obtained by deleting s distinct bits. For any two distinct words x, y: $|D(x) \cap D(y)| = 0$.

(n,s,m): ***(12,2,36)*** ***(13,2,55)*** ***(14,2,85)*** ***(15,2,132)*** ***(16,2,208)*** ***(13,3,16)*** ***(14,3,21)*** ***(15,3,29)*** ***(16,3,42)***

(c) Deletion Codes found to exist by CPro1, improving on results from Apr. 2025 article using FunSearch [37].

Table 1: **Main Results.** Combinatorial design problems by source, each with brief definition and a list of the open instances (in bold italic) solved by CPro1. For each of these open instances, the code generated by CPro1 constructed a verified solution. The Appendix includes solutions (Figs. 7 to 17) and full definitions as used by CPro1 (Tables 5 to 8).

obtain novel *Deletion Codes* [37]. Compared to FunSearch, CPro1 generates fewer candidates, allows open-ended strategies rather than just greedy functions, and omits FunSearch's iterative evolution.

In work concurrent with ours, AlphaEvolve [23] also built on FunSearch by allowing open-ended strategies. Compared to AlphaEvolve, CPro1 is distinguished by a uniform approach to automated hyperparameter tuning on candidate solutions [35], a focus on producing fast implementations in C with an explicit code optimization step [12, 25], and the application to combinatorial design problems.

The work presented here is also described in earlier preprints [27, 28].

2 Method

Selection of Combinatorial Designs (see Table 2): We select 16 types of combinatorial designs for which the 2006 *Handbook of Combinatorial Designs* [4] (henceforth *Handbook*) lists open instances, eliminating instances already solved in the literature [30, 22, 14, 9, 6, 13, 15, 7]. From recent (Feb. 2025) articles, we select those presenting open instances of combinatorial designs: this selects 2 of 6 articles from *Journal of Combinatorial Design*, and 2 of 447 arXiv math.CO combinatorics articles. We also include Cap Sets from the original FunSearch paper [26] and Deletion Codes from an April 2025 FunSearch paper [37].

For each selected problem, we provide a **textual definition**, a **Verifier in Python** that determines whether a proposed solution is correct, **Open Instances** for which existence is not yet known, and **Dev Instances** known to exist. Candidates are executed on Dev Instances for short amounts of compute time, with results checked by the Verifier, to identify the most promising approaches to try on the Open Instances using longer compute times.

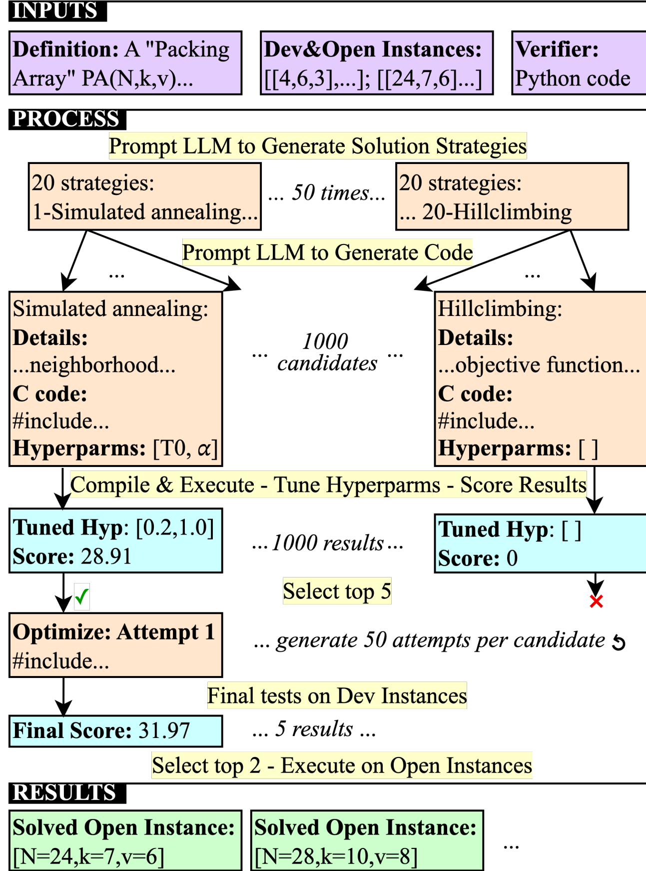

Figure 1: **Constructive Protocol CPro1**

Selected problems from the *Handbook*	Chapter
Balanced Incomplete Block Design	II.1
Packing Array (PA)	III.3
Orthogonal Array	III.6
Symmetric Weighing Matrix (SymmW)	V.2
Skew Weighing Matrix (SkewW)	V.2
Bhaskar Rao Design (BRD)	V.4
Balanced Ternary Design (BTD)	VI.2
Costas Array	VI.9
Covering Design	VI.11
Difference Triangle Set	VI.19
Perfect Mendelsohn Design	VI.35
Equidistant Permutation Array (EPA)	VI.44
Florentine Rectangle (FR)	VI.62
Circular Florentine Rectangle	VI.62
Tuscan-2 Square	VI.62
Supersimple Design	VI.57

Feb. 2025 problems	Source
Covering Sequence	[3]
Johnson Clique Cover	[18]
UNSQS	[2]
Covering Array	[29]

FunSearch Problems	Source
Cap Sets	[26]
Deletion Codes	[37]

Table 2: **Problem Selection**: 16 combinatorial designs with open instances from 2006 *Handbook of Combinatorial Designs* [4]; 4 recent problems from Feb. 2025 papers; 2 FunSearch problems

	PA	SymmW	SkewW	BTD	FR	EPA	BRD
o3-mini-high	Tabu	DFS	DFS	Tabu	GRASP	cSA	cSA,2p
- Reduce runtime	Tabu	DFS	DFS	Tabu		cSA	cSA,2p
- No final dev test	Tabu	DFS	DFS	Tabu			cSA,2p
- No optimization	Tabu	DFS	DFS	Tabu			2p
- No hyper tuning	Tabu	DFS	DFS	Tabu,RG			2p
GPT-4o	cSA	rSA	cSA	GA	DFS	cSA	
- Reduce runtime	cSA	rSA	cSA	GA	DFS		
- No final dev test	cSA	rSA	cSA	GA	DFS		
- No optimization	cSA	SA	cSA	GA			
- No hyper tuning	rSA	GA					
Mistral Large	cSA	cSA	cSA				
- Reduce runtime	cSA	cSA	cSA				
- No final dev test	cSA	cSA	cSA				
- No optimization	cSA		cSA				
- No hyper tuning	cSA						

Table 3: **Problem-Specific Results and Ablation.** Each row is an experiment, each column a type of combinatorial design from the *Handbook*. Colored cells show where the experiment solved at least one open instance: green indicates the experiment solved all the open instances that the initial run did, and yellow indicates an ablation experiment solved only a subset. The first set of rows use o3-mini-high, and the "-" rows stack up successive ablations: **Reduce runtime** reduces from 48 hours to 2 for open instances, **No final dev test** eliminates final 2 hour testing on dev instances (instead using original 50-second test results), **No optimization** eliminates the code optimization step, and **No hyper tuning** eliminates hyperparameter tuning (instead using defaults given by the LLM). The second set of rows use GPT-4o, and the third use Mistral Large. Each cell shows the strategies that obtained success: **DFS**: backtracking depth-first search, **GA**: genetic algorithm, **SA**: simulated annealing with slow cooling schedule, **rSA**: simulated annealing with periodic resets, **cSA**: constant-temperature simulated annealing, **Tabu**: Tabu search [10], **RG**: randomized greedy (randomized DFS within row, restart if any row fails), **2p**: 2-phase method for BRD: find binary incidence matrix, then add signs, **GRASP:** Greedy randomized adaptive search procedure [8].

Prototyping Set: We manually develop local search methods for 5 types of combinatorial designs, solving 1 open instance of EPA and 2 of SymmW; these two problems form our *prototyping set* for use during protocol development.

Protocol Development: In our protocol, an LLM takes a combinatorial design's definition as input and implements candidate solutions in C. The protocol executes the generated code on dev instances, checks results with the verifier, and selects the most promising candidates for use on open instances. We iteratively developed the protocol while testing only on the prototyping set, establishing protocol elements: prompting for diverse strategies separately from code generation [36], performing automated grid-based hyperparameter tuning for each candidate [35], testing candidates using multiple random seeds to encourage randomized strategies, and using an explicit code optimization step.

Fig. 1 outlines the final protocol CPro1. During initial testing, 1000 candidate programs are each scored on the basis of 50 seconds of execution using the tuned hyperparameters. Score is the number of instances solved, plus a tie-breaking bonus for faster solutions. The 5 top-scoring candidates proceed to optimization (also 50 seconds). For final testing on dev instances, the same 5 candidates (after optimization) are given 2 hours. The 2 top-scoring candidates from this then run for 48 hours on the open instances. The Appendix has details (Algorithm 1) and full prompts. Of several LLMs tested on the prototyping set, OpenAI o3-mini-high [24] performed best (Table 4), so the main results use it. We also report results using GPT-4o and the open-weights Mistral Large (Table 3).

3 Results

Table 1 shows the main results. CPro1 solves long-standing open instances for 7 types of combinatorial designs of the 16 selected from the *Handbook*. From recent combinatorial design literature, CPro1 improves upon Feb. 2025 results by solving open instances from 3 of the 4 selected papers. For

the Deletion Codes problem, FunSearch was reported to improve on state-of-the-art codes for 3 sets of parameters [37]. CPro1 finds further improvement for all 3, plus 6 other sets of parameters, substantially improving on the state of the art for small Deletion Codes. All of the open instance solutions, and the code that constructed them, are available on github. [1]

Ablation and Scaling: Ablation tests (Table 3) show all considered elements of the protocol are needed to obtain the full results. Scaled-down runs (Fig. 2) show success rates remain high at smaller scale for the o3-mini reasoning model, whereas GPT-4o performs relatively poorly.

Strategies Implemented by Generated Code: All successful strategies are randomized, and most appear well-optimized. Successful programs are 120-540 lines of C code, with o3-mini-high generating larger programs. Most use simulated annealing, genetic algorithms, tabu search, or depth-first search (Table 3). Each run has a total of 1000 candidates; common strategies are each proposed dozens of times during a run, giving adequate room to explore alternatives (e.g. various cost functions).

In general, the application of simulated annealing and tabu search to combinatorial designs is well established (*Handbook* chapter VII.6). However, from reviewing the literature, the application of these specifically to Balanced Ternary Designs, Bhaskar Rao Designs, and Equidistant Permutation Arrays appears to be novel. For Packing Arrays, CPro1's new solutions improve upon prior results obtained using simulated annealing [30] and SAT solvers [22].

For the UNSQS problem, CPro1's approach succeeds by addressing different constraints in two phases: first create an SQS using Knuth's Algorithm X [19], then perform tabu search for nested pairs. A two-phase approach also arises for Bhaskar Rao Designs: first find a 0/1 matrix that meets incidence constraints, then add +1/-1 signs. These two-phase approaches seem somewhat surprising; one might have anticipated that constraint interaction would prevent success. Prior two-phase approaches for the Covering Arrays problem [33, 34] have both phases addressing the same constraints.

Limitations: CPro1's positive results arise from automating computational experimentation that could have been done manually. The research community has only undertaken limited effort on such experimentation for some of the designs with positive results here; this may have left low-hanging fruit for CPro1. Some of the designs with no positive results here have received much greater attention. For example, CPro1 was unsuccessful on Covering Arrays, which have seen sustained development of progressively better computational techniques and mathematical constructions [32, 34].

The combinatorial design research community has greater focus on systematic mathematical constructions (e.g. using algebraic methods), rather than direct computational search for small designs. CPro1 doesn't necessarily exclude systematic constructions – for example, for Costas Arrays, some of the generated code implements systematic constructions such as the Welch method [11], solving larger dev instances than are solved by direct search. However, CPro1 was never able to extend these systematic constructions to solve open instances. The solutions produced by CPro1-generated heuristics generally do not have any obvious structure that could give insight into new systematic constructions.

CPro1 can only show existence by constructing a solution. For some of the open instances included here, there may exist no solution, but non-existence proofs are outside the scope of CPro1.

In work that appeared after this project was completed, other researchers also reported a UNSQS with $v = 14$ [20], and a Symmetric Weighing Matrix with $n = 22$ $w = 16$ [31], as we report here.

4 Conclusion

The protocol CPro1 uses LLMs to generate search heuristics, and successfully solves open instances of the existence problem for 7 types of combinatorial designs from the *Handbook of Combinatorial Designs*, 3 more from the Feb. 2025 combinatorial design literature, and also yields state of the art results for the Deletion Codes problem that was previously addressed by FunSearch. The code for CPro1 is available,[1] and the protocol can be run on additional types of combinatorial designs by supplying a textual definition, a Python verifier, and small collections of parameters for dev instances and open instances. For the task of showing existence of small open instances of combinatorial designs using heuristic computational search, LLMs with protocols like CPro1 are ready to contribute in an automated fashion.

[1] https://github.com/Constructive-Codes/CPro1

References

[1] Anthropic. Claude 3.5 Sonnet, 2024. https://www.anthropic.com/news/claude-3-5-sonnet.

[2] Y. M. Chee, S. H. Dau, T. Etzion, H. M. Kiah, and W. Zhang. Pairs in nested Steiner quadruple systems. *Journal of Combinatorial Designs*, 33(5):177–187, 2025.

[3] Y. M. Chee, T. Etzion, H. Ta, and V. K. Vu. Constructions of covering sequences and arrays, 2025. arXiv:2502.08424.

[4] C. J. Colbourn and J. H. Dinitz. *Handbook of combinatorial designs*. Taylor & Francis, 2006.

[5] DeepSeek. DeepSeek-V2.5, 2024. https://huggingface.co/deepseek-ai/DeepSeek-V2.5.

[6] J. Dinitz. New results in Part V, 2018. https://site.uvm.edu/jdinitz/?page_id=404.

[7] J. Dinitz. New results in Part VI, 2018. https://site.uvm.edu/jdinitz/?page_id=413.

[8] T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6:109–133, 1995.

[9] S. D. Georgiou, S. Stylianou, and H. Alrweili. On symmetric weighing matrices. *Mathematics*, 11:2076, 2023.

[10] F. Glover and M. Laguna. *Tabu search*. Springer, 1997.

[11] S. Golomb and H. Taylor. Constructions and properties of costas arrays. *Proceedings of the IEEE*, 72(9):1143–1163, 1984. doi: 10.1109/PROC.1984.12994.

[12] J. Gong, V. Voskanyan, P. Brookes, F. Wu, W. Jie, J. Xu, R. Giavrimis, M. Basios, L. Kanthan, and Z. Wang. Language models for code optimization: Survey, challenges and future directions, 2025. arXiv:2501.01277.

[13] D. Gordon. La Jolla Covering Repository Tables, 2025. https://ljcr.dmgordon.org/cover/table.html.

[14] M. Greig. Constructions using balanced n-ary designs. In *Designs*, pages 227–274. Springer, 2002.

[15] T. S. Griggs and A. R. Kozlik. The last two perfect Mendelsohn designs with block size 5. *Journal of Combinatorial Designs*, 28(12):865–868, 2020.

[16] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, 2025. arXiv:2501.12948.

[17] A. Hurst, A. Lerer, et al. GPT-4o system card, 2024. arXiv:2410.21276.

[18] S. F. Jørgensen. On the clique covering numbers of Johnson graphs, 2025. arXiv:2502.15019.

[19] D. E. Knuth. Dancing links, 2000. arXiv:cs/0011047.

[20] X.-N. Lu. Completely (quasi-)uniform nested boolean Steiner Quadruple Systems, 2025. arXiv:2509.06663.

[21] Mistral. Mistral Large 2, 2024. https://mistral.ai/news/mistral-large-2407.

[22] H. Noritake, M. Banbara, T. Soh, N. Tamura, and K. Inoue. Constraint modeling and SAT encoding of the packing array problem. *Computer Software*, 31:1_116–1_130, 2014.

[23] A. Novikov, N. Vu, M. Eisenberger, E. Dupont, et al. AlphaEvolve: A coding agent for scientific and algorithmic discovery, 2025. arXiv:2506.13131.

[24] OpenAI. OpenAI o3-mini system card, 2025. https://cdn.openai.com/o3-mini-system-card-feb10.pdf.

[25] R. Qiu, W. W. Zeng, H. Tong, J. Ezick, C. Lott, and H. Tong. How efficient is LLM-generated code? A rigorous & high-standard benchmark. *ICLR*, 2025.

[26] B. Romera-Paredes, M. Barekatain, A. Novikov, M. Balog, M. P. Kumar, E. Dupont, F. J. Ruiz, J. S. Ellenberg, P. Wang, O. Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

[27] C. D. Rosin. Using code generation to solve open instances of combinatorial design problems, January 2025. arXiv:2501.17725.

[28] C. D. Rosin. Using reasoning models to generate search heuristics that solve open instances of combinatorial design problems, May 2025. arXiv:2505.23881.

[29] K. Shokri and L. Moura. New families of strength-3 covering arrays using linear feedback shift register sequences. *Journal of Combinatorial Designs*, 2025.

[30] J. Stardom. Metaheuristics and the search for covering and packing arrays. Master's thesis, Simon Fraser University, 2001.

[31] S. N. Topno and S. Saurabh. A new symmetric weighing matrix SW(22,16). *Notes on Number Theory and Discrete Mathematics*, 31(3):443–447, 2025.

[32] J. Torres-Jimenez and E. Rodriguez-Tello. New bounds for binary covering arrays using simulated annealing. *Information Sciences*, 185(1):137–152, 2012.

[33] J. Torres-Jimenez, H. Avila-George, and I. Izquierdo-Marquez. A two-stage algorithm for combinatorial testing. *Optimization Letters*, 11(3):457–469, 2017.

[34] J. Torres-Jimenez, I. Izquierdo-Marquez, and H. Avila-George. Methods to construct uniform covering arrays. *IEEE Access*, 7:42774–42797, 2019.

[35] N. van Stein, D. Vermetten, and T. Bäck. In-the-loop hyper-parameter optimization for LLM-based automated design of heuristics. *ACM Transactions on Evolutionary Learning*, 2025. doi: 10.1145/3731567.

[36] E. Wang, F. Cassano, C. Wu, Y. Bai, W. Song, V. Nath, Z. Han, S. Hendryx, S. Yue, and H. Zhang. Planning in natural language improves LLM search for code generation. *ICLR*, 2025.

[37] F. Weindel and R. Heckel. LLM-guided search for deletion-correcting codes, 2025. arXiv:2504.00613.

[38] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, et al. Qwen2.5 technical report, 2024. arXiv:2412.15115.

A Appendix: Additional Tables and Figure, Protocol Pseudocode, Prompts, Definitions Used for Prompting, and Solutions to Open Instances

Table 4 and Figure 2 provide additional details and are included below. The protocol CPro1 that is outlined in Fig. 1 is specified in more detail in Algorithm 1.

The prompts that are used are shown in Figures 3 to 6. These are shown instantiated for the Packing Array (PA) combinatorial design problem; the text that can vary between problems is shown in italics and the rest of the prompt text is constant. Problem definitions, as used in prompting, are in Tables 5 to 8.

Figures 7 to 17 show verified designs that resolve open instances of combinatorial design problems as noted in Table 1. Each of these was constructed by code that was generated by protocol CPro1. One instance solution is provided here for each type of combinatorial design. Refer to the definitions in Tables 5 to 8 to interpret these solutions. The github repo for this paper contains the full set of solutions (https://github.com/Constructive-Codes/CPro1).

LLM	Ver. Date	#cand.	SymmW open solved	EPA open solved
GPT-4o [17]	2024-05-13	100	0	1
GPT-4o	2024-05-13	300	1	0
GPT-4o	2024-05-13	1000	2	1
Claude 3.5 Sonnet [1]	20241022	1000	2	0
Mistral Large [21]	2407	1000	2	0
DeepSeek v2.5 [5]	2024/09/05	1000	0	0
Qwen2.5-72B-Instruct [38]	11-2024	1000	0	0
o3-mini-high [24]	2025-01-31	1000	3	1
DeepSeek R1 [16]	2025-01-20	1000	2	1

Table 4: **Prototyping Set Results.** For each LLM and total number of candidate programs: the number of distinct open instances that are solved by CPro1 on each Prototyping Set problem. The *open solved* numbers are out of 4 open instances tested for SymmW, and 6 for EPA.

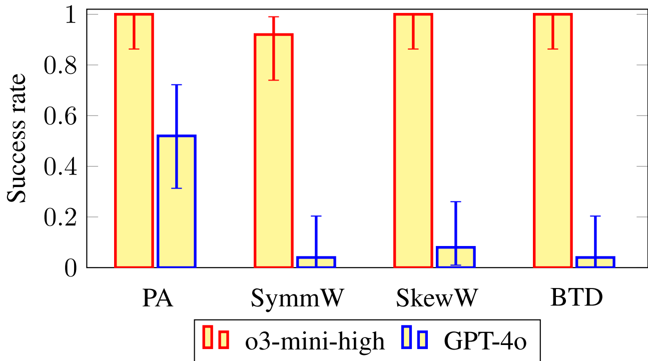

Figure 2: **Results from scaled-down 40-candidate runs.** With reduced runtime (2 hours) on open instances, no final dev test, and no optimization. Each bar shows the rate of successfully solving at least one open instance, across 25 repeated runs, with 95% Clopper-Pearson confidence interval. Each problem's difference between o3-mini-high and GPT-4o is significant (p<0.0001, Z-test).

Algorithm 1 Protocol **CPro1**. Note **prompt**(x) returns LLM result when prompted with x.

```
Input: Definition (of the problem), Dev & Open (instance parameter lists), Verifier (Python code)
Config for Code Generation: N = 20 strategies per rep, R = 50 reps
Config for Execution: Devseeds & Openseeds (so that #seeds * #instances = 32 parallel threads),
  Fulldevtime = 2 hours, Opentime = 48 hours (full run time limits)
Config for Hyperparameter Tuning: Init_gridsize = 1000, Init_runtime = 0.5 sec., Scale = 10
Config for Optimization: Opt_cand = 50, Opt_rounds = 5, Opt_delta = 0.1
Config for Selection: Devcand = 5 (for opt. & final Dev testing), Opencand = 2 (for Open instances)
Output: Verified designs for the Open instances (if found)

// Parallel sandboxed execution of candidate executable E for maximum time T:
def ex(E,HyperP,Inst,Seeds,T): return output & runtime of E with HyperP for T, on #Seeds × Inst

// Result scoring: number of verified correct results, plus speed bonus in the range of [0, 1]:
def sc(Results,T): return ∑_{r∈Results}[1 + (r's time/(T*|Results|))] if r passes Verifier check

// Hyperparameter tuning functions:
def one_grid(Min,Max,P): return coarse linear grid (1/3 of P), logarithmic/fine-grained to Min&Max

def hypergrid(Ranges): // Ranges has Min,Max,Default for each hyperparameter.
  // Allocate Init_gridsize points equally across |Ranges| + 1 grids as follows:
  G_balanced = cross product of one_grid(Min,Max,#points) for each hyperparameter
  G_i = cross product of one_grid for hyperparameter i with 3 vals {Min,Max,Default} for others
  return union of G_balanced and all G_i

def run_grid(E,Grid,T): return [Parms&sc(ex(E,Parms,Dev inst.,Devseeds,T),T) for Parms in Grid]

def hypertune(E,Grid,T):
  Results = run_grid(E,Grid,T)
  if |Grid| ≤ Scale: return single best-scoring Parms in Results
  else: return hypertune(E, (|Grid|/Scale) best Parms in Results, T*Scale) // smaller grid/more time
  // Final tests in hypertune always run for Init_runtime*Scale^((log_Scale Init_gridsize)−1) = 50 sec.

// Main protocol; Candidates is initialized to empty array
  for R(=50) reps do
    Strategies = prompt(Definition + "Please suggest " + N(=20) + " different approaches...")
    for S in Strategies do
      Details = prompt(Definition + "We have selected..." + S + "...Describe the elements...")
      Code,Hyperparm_ranges = prompt("Now implement...") //Continued chat; Details in context
      append Code,Hyperparm_ranges to Candidates
    end for
  end for  // We now have R*N=1000 candidates

  for C in Candidates do
    Compile C's source code to executable E
    Set C's Hyper_settings,Score = hypertune(E,hypergrid(C's Hyperparm_ranges),Init_runtime)
  end for
  truncate Candidates to the best Devcand according to their Score

  for C in Candidates do
    for i=1 to Opt_rounds(=5) do
      O = [prompt("...improve the performance..." + C + "...") * Opt_cand(=50) times]
      C' = highest-scoring single result of compiling, ex(), and sc() each of Opt_cand candidates
      if C' scores better than C by at least Opt_delta: replace C by C', else: break
    end for
  end for

  for C in Candidates do: Devsc = sc(ex(C's E,C's Hyper_settings,Dev inst,Devseeds,Fulldevtime))
  truncate Candidates to the best Opencand according to their Devsc

  for C in Candidates do
    Results = ex(C's executable E,C's Hyper_settings,Open instances,Openseeds,Opentime)
    output each design in Results that passes Verifier check
  end for
```

A "Packing Array" PA(N,k,v) is an N x k array (N rows and k columns), with each entry from the v-set {0,1,...v-1}, so that every N x 2 subarray contains every ordered pair of symbols at most once. Given (N,k,v), we want to construct PA(N,k,v). Please suggest 20 different approaches we could implement in C. For now, just describe the approaches. Then I will pick one of the approaches, and you will write the C code to test it. We will start testing on small parameters like *N=4 k=6 v=3*, and then once those work we will proceed to larger parameters like *N=32 k=5 v=6*. Format your list items like this example: "12. **Strategy Name**: Sentences describing strategy, all on one line..."

Figure 3: Strategies prompt to the LLM, instantiated (italicized text) for Packing Arrays.

A "Packing Array" PA(N,k,v) is an N x k array (N rows and k columns), with each entry from the v-set {0,1,...v-1}, so that every N x 2 subarray contains every ordered pair of symbols at most once. Given (N,k,v), we want to construct PA(N,k,v).

We have selected the following approach:
Simulated Annealing: Use simulated annealing to slowly improve a randomly initialized array by making small changes and accepting them based on a cooling schedule. Do not terminate until a valid solution is found.

Describe the elements of this approach to constructing a *Packing Array*. Do not yet write code; just describe the details of the approach.

Figure 4: Details prompt to the LLM, instantiated (italicized text) for Packing Arrays with a Simulated Annealing strategy resulting from the prompt in Fig. 3. Note "Do not terminate until a valid solution is found" is not italicized and is a fixed part of the template that is always appended to the strategy description generated by the LLM.

Now implement this approach in C, following in detail the plan described above. Provide the complete code. The code should only print out the final *Packing Array* once a valid solution is found.

I will be running the code from the Linux command line. Please have the C code take command-line parameters: *N k v* seed (in that order), followed by additional parameters as needed which represent hyperparameters of your approach. The seed is the random seed (if no random seed is needed, still accept this parameter but ignore it).

After giving the complete code, for each hyperparameter that is an extra command-line parameter, provide a specification in JSON with fields "name","min","max","default" specifying the name, minimum value, maximum value, and default value for the hyperparameter. For example: "name":"gamma", "min":0.0, "max":2.0, "default":0.5. If no hyperparameters are required then just state "No Hyperparameters Required" after giving the complete code. I will be using Linux timeout to set a time limit on execution of your program, and for challenging *PA* parameters this will be a long timeout (hours). So to maximize chances of finding a solution your code should keep running indefinitely until it finds a valid solution. Therefore, eliminate hyperparameters that would control termination, since your program needn't terminate until it succeeds.

We will start testing with small problem parameters like *N=4 k=6 v=3*. Once those work, we can then test further refinements and move towards larger problem parameters like *N=32 k=5 v=6*.

Figure 5: Code generation prompt to the LLM, instantiated (italicized text) for Packing Arrays. Note this continues the chat following on from the Details prompt in Fig. 4; so the Details prompt and response are in context when the LLM responds to this prompt.

A "Packing Array" PA(N,k,v) is an N x k array (N rows and k columns), with each entry from the v-set {0,1,...v-1}, so that every N x 2 subarray contains every ordered pair of symbols at most once. Given (N,k,v), we want to construct PA(N,k,v).

We have selected the following approach:
Simulated Annealing: Use simulated annealing to slowly improve a randomly initialized array by making small changes and accepting them based on a cooling schedule. Do not terminate until a valid solution is found.

The C code below implements this approach. It takes command-line parameters: *N k v* seed (in that order) where seed is the random number generator seed, followed by additional parameters which represent hyperparameters of the approach.

We are seeking to make one small change to significantly improve the performance of this code.

```
#include <stdio.h>
#include <stdlib.h>
...
```

With these hyperparameters chosen by hyperparameter tuning: *T0=0.1575068233468584, cooling_rate=1.0*. This code solves *N=4 k=6 v=3* in an average of *0.0026* seconds across *2* attempts, solves *N=6 k=5 v=3* in an average of *0.0028* seconds across *2* attempts, solves *N=9 k=6 v=4* in an average of *0.0686* seconds across *2* attempts, solves *N=20 k=6 v=5* in an average of *0.0789* seconds across *2* attempts, solves *N=25 k=6 v=5* in an average of *0.0306* seconds across *2* attempts, solves *N=25 k=5 v=6* in an average of *0.0139* seconds across *2* attempts, solves *N=30 k=5 v=6* in an average of *0.9323* seconds across *2* attempts, solves *N=31 k=5 v=6* in *1* out of *2* attempts with a time limit of 50.0 seconds each, solves *N=16 k=7 v=6* in an average of *0.011* seconds across *2* attempts, solves *N=23 k=7 v=6* in *1* out of *2* attempts with a time limit of 50.0 seconds each, solves *N=17 k=10 v=8* in an average of *0.014* seconds across *2* attempts, solves *N=22 k=10 v=8* in an average of *0.1002* seconds across *2* attempts, solves *N=25 k=10 v=8* in an average of *6.9132* seconds across *2* attempts, solves *N=14 k=11 v=8* in an average of *0.0095* seconds across *2* attempts, solves *N=19 k=11 v=8* in an average of *0.0396* seconds across *2* attempts, and solves *N=22 k=11 v=8* in an average of *4.2044* seconds across *2* attempts. We want to make one small change to the code to significantly improve performance, without parallelizing or changing the algorithm or the hyperparameter handling. Describe your plan for making one small change to significantly improve performance. Then, implement your plan and provide the complete updated code.

Figure 6: Optimization prompt to the LLM for generating candidate optimized code, instantiated (italicized text) for a Packing Arrays run with a Simulated Annealing strategy resulting from the prompt in Fig. 3, code (shown in abbreviated form) resulting from the prompt in Fig. 5, and results from automated hyperparameter tuning.

Definitions
A **"Balanced Incomplete Block Design"** BIBD(v,b,r,k,L) is a pair (V,B) where V is a v-set and B is a collection of b k-subsets of V (blocks) such that each element of V is contained in exactly r blocks and any 2-subset of V is contained in exactly L blocks. The BIBD is represented by a v by b incidence matrix (v rows and b columns) with elements in 0,1. The matrix element m_{ij} in the i'th row and j'th column is 1 iff element i is contained in block j. The sum of each row is r, and the sum of each column is k. For each pair of distinct elements y and z, sum_{j=1}^{b} m_{yj} m_{zj} = L. Given (v,b,r,k,L) we want to find the incidence matrix for a valid BIBD(v,b,r,k,L).
A **"Packing Array"** PA(N,k,v) is an N x k array (N rows and k columns), with each entry from the v-set 0,1,...v-1, so that every N x 2 subarray contains every ordered pair of symbols at most once. Given (N,k,v), we want to construct PA(N,k,v).
An **"Orthogonal Array"** OA(N,k,s) of size N, degree k, and order s, is a k x N array (k rows and N columns) with entries from the s-set {0,1,...,s-1} having the property that in every 2 x N submatrix, every 2 x 1 column vector appears the same number (lambda) of times. lambda is called the index of the OA, and lambda = N/s^2. Given (N,k,s,lambda), we want to construct an OA(N,k,s) with index lambda.
A "weighing matrix" W(n,w) with parameters (n,w) is an n by n square matrix (n rows and n columns) with entries in {0,1,-1} that satisfies W W^T = wI. That is, W times its transpose is equal to the constant w times the identity matrix I. The weighing matrix will have w nonzero entries in each row and each column. And each pair of distinct rows is orthogonal (dot product zero). Given (n,w), we want to construct "SymmW", a **symmetric weighing matrix** W(n,w) that satisfies these properties and is also a symmetric matrix.
A "weighing matrix" W(n,w) with parameters (n,w) is an n by n square matrix (n rows and n columns) with entries in {0,1,-1} that satisfies W W^T = wI. That is, W times its transpose is equal to the constant w times the identity matrix I. The weighing matrix will have w nonzero entries in each row and each column. And each pair of distinct rows is orthogonal (dot product zero). Given (n,w), we want to construct "SkewW", a **skew weighing matrix** W(n,w) that satisfies these properties and is also a skew matrix: that is, W^T = -W.
A **"Bhaskar Rao Design"** BRD(v,b,r,k,L) is represented by a v by b array (v rows and b columns) with elements a_{ij} in {-1,0,1}. Each row contains exactly r nonzero elements, and each column contains exactly k nonzero elements. For any pair of distinct rows, the list of pairwise element products must contain -1 L/2 times, and must contain +1 L/2 times. That is, for distinct rows f and g, the set of pairwise element products {a_{fj}*a_{gj}} contains -1 L/2 times, +1 L/2 times, and 0 b-L times. Given (v,b,r,k,L) we want to find a valid BRD(v,b,r,k,L).
A **"Balanced Ternary Design"** BTD(V,B;p1,p2,R;K,L) is an arrangement of V elements into B multisets, or blocks, each of cardinality K (K<=V) satisfying: 1. Each element appears R=p1 + 2*p2 times altogether, with multiplicity one in exactly p1 blocks and multiplicity two in exactly p2 blocks. 2. Every pair of distinct elements appears L times; that is, if m_{vb} is the multiplicity of the v'th element in the b'th block, then for every pair of distinct elements v and w, sum_{b=1}^{B} m_{vb} m_{wb} = L. The BTD is represented by a V by B incidence matrix with elements in 0,1,2. The matrix element m_{vb} in the v'th row and b'th column is the multiplicity of the v'th element in the b'th block. The sum of each row is R, and the sum of each column is K. Given (V,B,p1,p2,R,K,L) we want to find BTD(V,B;p1,p2,R;K,L).
A **"Costas Array"** of order n is an n by n array of dots and blanks that satisfies: (1) There are n dots and n(n-1) blanks, with exactly one dot in each row and each column. (2) All the segments between pairs of dots differ in length or slope. We will represent the Costas Array by a one-dimensional list "CA" of length n, that contains a permutation of {0,1,...,n-1}. CA[i] identifies the row for the dot that is in column i of the grid. Condition (1) is automatically satisfied by this representation. Condition (2) is satisfied if the tuples (j-i,CA[j]-CA[i]) are unique across all j>i with 0<=i,j<n. Given n, we want to construct the one-dimensional list CA that represents a valid Costas Array of order n.

Table 5: **Definitions for use in prompting**: *Handbook* combinatorial design problems part 1. Note the definitions use ASCII symbols for the sake of prompting.

A **"Covering"** Cov(t,v,k,n) is a pair (X,B), where X is a v-set of elements and B is a collection of k-subsets of X, such that every t-subset of X occurs in at least one block in B. B has n blocks, and it is required that t<k. We represent the Covering by an n by v incidence matrix (n rows and v columns) with elements in {0,1}; a 1 in row i column j indicates that block i contains the j'th element. There are k 1's per row. Given (t,v,k,n) we want to construct a Cov(t,v,k,n) and provide the incidence matrix.
A **"Difference Triangle Set"** (n,k)-DTS is a set X={X_1,...,X_n} where for 1<=i<=n, X_i={a_{i0}, a_{i1},..., a_{ik}} with a_{ij} an integer and with 0 = a_{i0} < a_{i1} <a_{i2} < ... < a_{ik}. The differences a_{il}-a_{ij} for 1<=i<=n, 0<=j!=l<=k are all distinct and nonzero. The "scope" of a DTS is the max of all a_{ij} in the DTS. Given (n,k) and s, we want to construct an (n,k)-DTS with scope s. The (n,k)-DTS should be represented by an n by k+1 array (n rows and k+1 columns) of elements a_{ij} with 1<=i<=n and 0<=j<=k.
Given a k-tuple (x_0, x_1, x_2, ..., x_{k-1}), elements x_i, x_{i+t} are t-apart in the k-tuple, where i+t is taken modulo k. A **"Perfect Mendelsohn Design"** with parameters v and k is denoted as a (v,k)-PMD. It is a set V={0,1,...,v-1} of size v together with a collection B of blocks of ordered k-tuples of distinct elements from V, such that for every i=1,2,3,...,k-1 each ordered pair (x,y) of distinct elements from V is i-apart in exactly one block. Note since there are v*(v-1) pairs of distinct elements that must be 1-apart in exactly one block, and each block has k pairs that are 1-apart, the design will contain b=v*(v-1)/k blocks. We will use b to denote the number of blocks. Given (v,k,b), we want to construct a (v,k)-PMD with b blocks. We will represent the PMD with a b by k array (b rows and k columns), with each element of the array chosen from {0,1,...,v-1}.
An **"equidistant permutation array"** (EPA) with parameters (n,d,m) can be represented as an m by n matrix (m rows and n columns), where each row is the permutation of the numbers 0 to n-1. Each pair of distinct rows must differ in exactly d positions. Given (n,d,m), we want to construct an equidistant permutation array (EPA) with these parameters.
A **"Florentine Rectangle"** FR(r,n) is an r x n array (r rows and n columns), with each row having a permutation of the set of symbols S={0,1,2,...,n-1}, such that for any two distinct symbols a and b in S and each m in {1,2,3,...,n-1} there is at most one row in which b appears in the position which is m steps to the right of a. A single row will have n-m pairs of symbols a,b with b being m steps to the right of a; so n-1 pairs with b directly to the right of a, n-2 with b 2 steps to the right of a, and only 1 pair with b n-1 steps to the right of a. Given (r,n) we want to construct a FR(r,n).
A **"Circular Florentine Rectangle"** CFR(r,n) is an r x n array (r rows and n columns), with each row having a permutation of the set of symbols S={0,1,2,...,n-1}, such that for any two distinct symbols a and b in S and each m in {1,2,3,...,n-1} there is at most one row in which b appears in the position which is m steps to the right of a. "Steps to the right" is taken circularly - so if a is at position i then b is at position (i+m) mod n. Given (r,n) we want to construct a CFR(r,n).
A **"Tuscan-2 Square"** T2(n) of size n is an n x n array (n rows and n columns), with each row having a permutation of the set of symbols S={0,1,2,...,n-1}, such that any two distinct symbols a and b in S have exactly one row in which b appears in the position directly to the right of a, and at most one row in which b appears two positions to the right of a (with one symbol between). Given n, we want to construct a T2(n).
A **"Supersimple Balanced Incomplete Block Design"** SBIBD(v,b,r,k,L) is a pair (V,B) where V is a v-set and B is a collection of b k-subsets of V (blocks) such that each element of V is contained in exactly r blocks, any 2-subset of V is contained in exactly L blocks, and any two distinct blocks have at most two elements in common. The SBIBD is represented by a v by b incidence matrix (v rows and b columns) with elements in {0,1}. The matrix element m_{ij} in the i'th row and j'th column is 1 iff element i is contained in block j. The sum of each row is r, and the sum of each column is k. For each pair of distinct elements y and z, sum_{j=1}^{b} m_{yj} m_{zj} = L. For each pair of distinct blocks g and h, sum_{i=1}^{v} m_{ig} m_{ih} <= 2. Given (v,b,r,k,L) we want to find the incidence matrix for a valid SBIBD(v,b,r,k,L).

Table 6: **Definitions for use in prompting**: *Handbook* combinatorial design problems part 2. Note the definitions use ASCII symbols for the sake of prompting.

An "(n,R)-**Covering Sequence**" (abbreviated "(n,R)-CS") of length L is a cyclic sequence x_0,x_1,...,x_{L-1} of length L, over the binary alphabet (x_i is in {0,1} for all j) such that for any length-n binary word y_0,y_1,...,y_{n-1} there exists a j such that subsequence x_j,x_{(j+1) mod L},x_{(j+2) mod L}...,x_{(j+n-1) mod L} of length n is Hamming distance at most R away from the word. That is, y_0,...,y_{n-1} and x_j,...,x_{(j+n-1) mod L} differ in at most R positions. For our purposes, n <= 16 and R <= 3 and L <= 1200. Given (n,R,L) we want to construct a (n,R)-CS of length at most L. Output (n,R)-CS as a list of values in {0,1} separated by spaces, all on one line.
The Johnson Graph J(N,k) with k<=N/2 is the graph whose vertices are k-element subsets of [N]={1,2,...,N}, with two subsets connected by an edge if their intersection has size exactly k-1. A **"Johnson Clique Cover"** JCC(N,k,C) of size C is a set of C cliques in J(N,k), such that the union of these cliques includes all vertices in J(N,k). Note the cliques in the clique cover do not need to be disjoint; they may share vertices. For our purposes, N<=15 and C<800. Given (N,k,C) we want to construct a JCC(N,k,C). It is a theorem that it suffices to consider clique covers that consist only of maximal cliques, and that all maximal cliques of J(N,k) are either type 0 or type 1, defined as follows. A type 0 clique specifies a k-1 element subset S of [N], and consists of all vertices corresponding to the subset S plus x, for each x that is an element of [N] that is not in S. A type 1 clique specifies k+1 elements in [N], and consists of all vertices corresponding to the subset S excluding x, for each x that is an element of S. Specify a clique as a space-separated list of integers where the first integer is the type (0 or 1) and the remaining integers specify the elements of S. For example in J(5,2) the type 0 clique "0 1" consists of vertices for the subsets 1,2, 1,3, 1,4, and 1,5; and the type 1 clique "1 3 4 5" consists of vertices for the subsets {4,5}, {3,4}, and {3,5}. Output the clique cover as C lines, with one clique per line, where each line is a space-separated list of integers starting with the type (0 or 1).
A "Steiner Quadruple System" (or SQS) of order v consists of a set of blocks, with each block containing 4 elements of the set V={0,1,...,v-1}, such that each subset of 3 elements of V is contained in exactly one block. There are v*(v-1)*(v-2)/6 subsets of 3 elements of V, and each block covers 4 of these subsets, so the SQS will have v*(v-1)*(v-2)/24 blocks. A **"Uniform Nested Steiner Quadruple System"** of order v splits each block into two "ND-pairs" of two elements each, such that each distinct ND-pair appears the same number of times. Set p denote the number of distinct ND-pairs that appear among the blocks; it may be that each of the v*(v-1)/2 subsets of 2 elements from V appears as an ND-pair so that p = v*(v-1)/2, or it may be that some subsets of 2 elements from V don't appear as an ND-pair and then we have p < v*(v-1)/2. There are v*(v-1)*(v-2)/12 ND-pairs, so each of the p distinct ND-pairs appears v*(v-1)*(v-2)/(12*p) times. We call such a design a UNSQS(v,p). Given (v,p) we want to construct a UNSQS(v,p). For our purposes, 8<=v<=58 and 28<=p<=1260. Output the UNSQS(v,p) as v*(v-1)*(v-2)/24 lines, one block per line, with each line having a space-separated list of 4 numbers identifying the elements of V in the block. The first ND-pair in a block should be the first two elements listed for the block, and the second ND-pair is the last two elements listed. Order does not matter within an ND-pair.
A "**Covering Array** of Strength 3" CA3(N,k,v) is an N x k array (N rows and k columns), with each entry from the v-set of symbols {0,1,...v-1}, so that every N x 3 subarray contains every ordered triple of symbols at least once. Given (N,k,v), we want to construct CA3(N,k,v). For our purposes, N<1000, k<1000, and v<6. Output the CA3(N,k,v) as N lines with one row per line, with each line a space-separated list of k columns where each column is an integer in {0,1,...,v-1}.

Table 7: **Definitions for use in prompting**: Feb. 2025 combinatorial design problems. Note the definitions use ASCII symbols for the sake of prompting.

A **"Cap Set"** CS(n,s) is a subset S of Z_3^{n}, such that S has at least s distinct points, and no three points {x,y,z} in S satisfy x+y+z=0 (vector addition over Z_3^{n}, so addition is taken modulo 3). We represent the cap set by an array with at least s rows, and n columns in each row. Elements of the array are from Z_3={0,1,2}, and each row represents a point in Z_3^{n} which is an element of S. Given (n,s) we want to construct a Cap Set CS(n,s).
A **"Deletion Code"** DC(n,s,m) with parameters (n,s,m) is a set m binary words of length n, such that any two distinct words from the set do not share any any length n-s word obtained by deleting s bits from each of the two words. Given a length n word x, let D(x) be the set of length n-s words obtained from x by deleting two distinct bits (at positions which need not be adjacent). Note D(x) has (n choose s) members. Our requirement is that, for any two distinct words x and y in DC(n,s,m), D(x) and D(y) have the empty intersection. Given (n,s,m) we want to construct a DC(n,s,m). For our purposes, 7<=n<=16, s is 2 or 3, and m<250. Output the DC(n,s,m) as an m by n array, where each row is a space-separated list of n bits in {0,1} representing one word in the Deletion Code.

Table 8: **Definitions for use in prompting**: FunSearch problems. Note the definitions use ASCII symbols for the sake of prompting.

1	3	2	5	5	2	1
5	1	3	1	0	5	1
4	2	0	1	2	2	4
0	2	1	4	5	5	0
2	4	0	5	1	1	0
2	5	5	2	0	2	5
5	0	1	3	1	2	2
1	5	3	3	2	0	0
3	4	3	2	5	3	4
4	1	1	2	3	1	3
2	2	2	3	4	3	3
1	0	5	0	3	5	4
4	4	2	0	0	0	2
4	5	4	4	1	3	1
2	3	3	4	3	4	2
1	4	1	1	4	4	5
3	1	4	0	4	2	0
3	3	5	1	1	0	3
5	5	0	0	5	4	3
0	1	5	5	2	3	2
3	0	2	4	2	1	5
0	3	4	3	0	1	4
5	2	4	5	3	0	5
0	0	0	2	4	0	1

Figure 7: Packing Array with N=24 k=7 v=6

0	0	0	−1	−1	−1	−1	0	0	0	1	−1	0	−1	−1	−1	0	0	0
0	1	0	1	−1	0	0	−1	0	1	0	−1	−1	1	0	0	−1	0	0
0	0	1	−1	0	−1	1	0	−1	1	−1	0	−1	0	0	0	1	0	0
−1	1	−1	1	−1	0	0	1	0	0	−1	0	0	−1	0	0	1	0	0
−1	−1	0	−1	−1	0	−1	0	0	0	−1	0	0	0	1	1	−1	0	0
−1	0	−1	0	0	−1	1	0	−1	0	1	1	0	0	0	0	−1	1	0
−1	0	1	0	−1	1	0	0	−1	0	1	0	1	1	0	0	1	0	0
0	−1	0	1	0	0	0	−1	−1	0	0	0	0	−1	−1	1	0	−1	−1
0	0	−1	0	0	−1	−1	−1	0	0	0	1	0	1	0	0	1	−1	1
0	1	1	0	0	0	0	0	0	1	0	1	1	−1	0	0	−1	−1	1
1	0	−1	−1	−1	1	1	0	0	0	0	0	0	0	−1	1	0	0	1
−1	−1	0	0	0	1	0	0	1	1	0	1	−1	0	−1	−1	0	0	0
0	−1	−1	0	0	0	1	0	0	1	0	−1	1	0	1	−1	0	−1	0
−1	1	0	−1	0	0	1	−1	1	−1	0	0	0	0	0	0	0	−1	−1
−1	0	0	0	1	0	0	−1	0	0	−1	−1	1	0	−1	0	0	1	1
−1	0	0	0	1	0	0	1	0	0	1	−1	−1	0	0	1	0	−1	1
0	−1	1	1	−1	−1	1	0	1	−1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	1	0	−1	−1	−1	0	0	−1	−1	1	−1	0	0	1
0	0	0	0	0	0	0	−1	1	1	1	0	0	−1	1	1	1	1	0

Figure 8: Symmetric Weighing Matrix with n=19 w=9

$$
\begin{array}{|rrrrrrrrrrrrrrrrrr|}
\hline
0 & 0 & -1 & 0 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 1 & 1 & -1 & 1 & 1 & 0 & 0 \\
0 & 0 & 1 & -1 & 0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & -1 \\
1 & -1 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 1 \\
0 & 1 & 0 & 0 & -1 & 1 & 1 & 0 & 0 & -1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & 1 & -1 & -1 & 0 & 0 & 1 & 1 & 0 \\
1 & -1 & 0 & -1 & 0 & 0 & 1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\
1 & 1 & 1 & -1 & 0 & -1 & 0 & 0 & -1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 1 & -1 & -1 & 0 & 0 & 0 & 1 & 0 & 1 \\
1 & 0 & -1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 1 & -1 \\
0 & 0 & 1 & 1 & 1 & 1 & 0 & -1 & 0 & 0 & 0 & -1 & 1 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & -1 & 1 & 1 & 1 & 0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & -1 \\
-1 & 0 & 0 & -1 & 1 & 0 & 1 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\
-1 & -1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & -1 & 0 & 1 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 1 & 0 & 1 & 0 & -1 & -1 & 1 & 0 \\
-1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & -1 & 1 & 1 & 0 & -1 & -1 & 0 \\
-1 & -1 & 1 & 0 & -1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & -1 & 1 & -1 & 0 & -1 \\
0 & 1 & -1 & -1 & 0 & 0 & -1 & -1 & 1 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\
\hline
\end{array}
$$

Figure 9: Skew Weighing Matrix with n=18 w=9

$$
\begin{array}{rrrrrrrrrrrrrrr}
1 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & -1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 0 \\
0 & -1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 \\
0 & 1 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & -1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 & 1 & 0 \\
0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & -1 & -1 \\
1 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 \\
1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & 0 & -1 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 1 & 0 & 0 & 0 \\
-1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & -1 & -1 & 0 & 0 \\
1 & 0 & 0 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
1 & 0 & 1 & -1 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & -1 & 0 & -1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 & -1 & 1 \\
0 & 0 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\
-1 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & -1 & 0 & 0 & 0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & -1 & 0 & 0 \\
-1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 1 & 1 & 0 & -1 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & -1 & -1 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 1 & -1 & 0 & -1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & -1 \\
1 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\
0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 \\
1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 & 0 & -1 & 0 & 1 & 0 & -1 & 0
\end{array}
$$

Figure 10: Bhaskar Rao Design with parameters (15,42,14,5,4). Note the transpose is shown.

2	1	1	0	1	0	0	1	0	0	0	1	1	0	1	2	1
0	2	1	1	0	0	1	0	1	1	1	1	0	0	2	1	0
1	0	0	0	0	2	0	1	2	1	1	1	1	0	1	1	0
1	1	1	0	0	0	1	2	1	0	2	1	0	1	0	0	1
0	0	1	0	1	1	2	0	0	1	1	2	1	0	0	1	1
0	1	2	0	1	1	1	1	1	0	0	0	2	1	1	0	0
1	0	1	2	1	1	1	2	0	1	0	1	0	0	1	0	0
0	1	0	1	1	1	1	1	1	0	0	1	0	2	0	2	0
0	2	0	1	1	1	0	1	1	1	0	1	1	0	0	0	2
1	1	2	1	0	2	0	0	0	1	1	0	0	1	0	1	1
1	0	1	1	0	0	0	0	1	1	0	2	1	2	1	0	1
1	0	1	2	1	0	1	0	2	0	1	0	1	0	0	1	1
1	1	0	1	2	1	0	0	0	0	2	1	1	1	1	0	0
1	1	0	1	0	0	1	1	0	2	1	0	2	1	0	1	0
0	0	0	1	0	1	1	1	0	0	1	0	1	1	2	1	2
2	1	0	0	1	1	2	0	1	1	0	0	0	1	1	0	1
0	0	1	0	2	0	0	1	1	2	1	0	0	1	1	1	1

Figure 11: Balanced Ternary Design with parameters (17,17;8,2,12;12,8)

6	5	7	9	1	4	11	2	10	8	0	3
0	9	2	8	1	4	11	3	10	6	7	5
4	5	7	8	1	0	6	3	10	2	9	11
6	7	3	8	1	0	11	9	10	5	2	4
6	5	0	8	9	7	11	3	10	1	4	2
6	5	2	8	1	9	7	11	10	4	3	0
4	5	3	0	1	9	11	7	10	6	8	2
0	5	4	6	1	7	11	9	10	2	3	8
7	5	1	8	4	0	11	2	10	6	3	9
4	6	9	8	1	7	11	2	10	3	5	0
0	5	7	8	6	9	11	4	10	3	2	1
3	5	7	8	1	10	11	9	2	6	4	0
7	5	8	4	1	6	11	3	10	9	2	0
4	5	2	8	0	6	11	9	10	7	1	3
9	5	3	8	7	4	11	1	10	2	6	0
7	5	6	8	1	4	0	9	10	3	11	2
0	5	3	8	1	6	9	2	10	11	4	7
9	5	2	7	1	0	11	8	10	3	4	6
9	5	11	8	1	7	4	0	10	6	2	3
7	0	5	8	1	9	11	6	10	2	4	3
9	4	7	8	1	6	11	5	10	0	3	2

Figure 12: Equidistant Permutation Array with n=12 d=8 m=21

8	2	9	16	15	18	19	10	1	4	6	3	17	7	12	13	5	14	0	11
11	3	19	1	2	18	8	7	14	10	16	17	9	13	4	0	15	5	12	6
13	11	4	3	2	16	9	0	6	8	5	19	7	10	12	15	1	18	14	17
19	13	12	16	2	8	1	5	15	14	3	0	18	17	11	7	6	9	4	10
8	6	17	14	5	9	12	0	7	16	4	13	18	1	19	3	15	11	10	2
12	11	0	13	10	7	1	15	17	2	4	5	6	19	16	18	3	14	9	8
18	10	3	5	0	4	15	13	9	11	14	1	7	8	16	12	19	17	6	2

Figure 13: Florentine Rectangle with r=7 n=20

01011101010011100001000000110110011001101101000101011000111101111110010

Figure 14: Covering Sequence with n=9 R=1 L=71

0	2	3	4		
1	1	3	4	9	13
0	3	9	12		
0	1	2	3		
1	1	3	5	6	11
0	5	7	10		
0	2	8	9		
0	2	5	12		
0	6	7	8		
0	3	6	13		
0	2	6	9		
0	2	9	13		
0	1	11	13		
0	1	6	8		
0	9	10	11		
0	1	4	6		
1	5	9	10	12	13
0	3	6	10		
1	1	2	6	10	11
0	3	10	13		
1	1	6	7	9	13
1	6	8	9	12	13
1	4	5	6	11	12
1	3	7	9	11	13
0	6	7	11		
0	2	8	12		
0	1	2	7		
1	3	5	7	11	12
0	4	5	9		
0	4	7	12		
0	3	8	13		
0	6	8	11		
0	5	7	9		
0	2	8	10		
0	6	9	12		
0	1	4	10		
0	6	10	13		
1	2	5	7	11	13
0	4	7	9		
0	2	5	6		
1	1	5	6	7	12
0	11	12	13		
0	7	12	13		
1	1	3	8	10	11
1	1	2	4	9	12
0	2	6	13		
0	7	8	11		
0	8	10	13		
0	4	7	13		
1	2	7	9	11	12
0	1	3	7		
0	3	5	8		
0	2	10	12		

0	1	9	11		
0	3	8	9		
0	8	10	12		
0	1	7	10		
0	1	5	9		
1	2	3	5	9	10
0	3	5	13		
0	3	10	12		
0	2	10	13		
0	1	12	13		
0	7	10	11		
0	4	12	13		
1	2	3	6	8	9
1	5	6	8	9	10
1	4	5	8	11	13
0	1	5	10		
0	2	6	12		
0	2	4	11		
0	4	9	10		
1	3	4	7	8	10
1	2	4	5	7	8
0	3	6	9		
1	3	4	5	6	7
0	7	9	10		
0	2	8	13		
1	1	7	8	9	12
1	4	8	9	11	12
0	6	10	12		
1	1	5	7	8	13
0	2	3	7		
0	8	9	13		
1	1	3	4	5	12
0	3	8	12		
0	5	9	11		
0	1	11	12		
0	1	4	8		
0	1	9	10		
1	4	6	9	11	13
0	3	4	11		
0	2	3	11		
0	4	6	8		
1	1	2	5	8	11
1	2	3	9	12	13
0	3	6	12		
0	4	10	11		
0	5	10	11		
0	4	5	10		
1	1	4	5	7	11
0	5	8	12		
1	1	2	4	5	13
1	2	4	6	7	10
0	5	6	13		

Figure 15: Johnson Clique Cover with N=13 k=4 C=105

0	1	2	3
0	4	1	5
0	6	1	7
0	9	1	8
0	10	1	11
0	1	12	13
0	4	2	6
0	7	2	5
0	10	2	8
0	2	9	12
0	2	11	13
0	8	3	4
0	5	3	9
0	3	6	10
0	3	7	13
0	11	3	12
0	11	4	7
0	13	4	9
0	12	4	10
0	5	6	12
0	8	5	11
0	13	5	10
0	6	8	13
0	9	6	11
0	12	7	8
0	7	9	10
1	4	2	7
1	2	5	6
1	2	8	11
1	13	2	9
1	10	2	12
1	4	3	10
1	3	5	12
1	9	3	6
1	3	7	11
1	8	3	13
1	6	4	13
1	12	4	8
1	9	4	11
1	7	5	8
1	5	9	10
1	11	5	13
1	10	6	8
1	6	11	12
1	12	7	9
1	13	7	10
2	13	3	4
2	8	3	5
2	11	3	6
2	3	7	12
2	10	3	9
2	4	5	10
2	4	8	9
2	11	4	12
2	5	9	11
2	12	5	13
2	9	6	7
2	6	8	12
2	10	6	13
2	13	7	8
2	7	10	11
3	11	4	5
3	12	4	6
3	7	4	9
3	13	5	6
3	5	7	10
3	7	6	8
3	8	9	12
3	8	10	11
3	11	9	13
3	10	12	13
4	6	5	7
4	5	8	13
4	12	5	9
4	8	6	11
4	10	6	9
4	7	8	10
4	13	7	12
4	11	10	13
5	8	6	9
5	11	6	10
5	9	7	13
5	7	11	12
5	12	8	10
6	7	10	12
6	13	7	11
6	12	9	13
7	9	8	11
8	9	10	13
8	12	11	13
9	11	10	12

Figure 16: Uniform Nested Steiner Quadruple System with v=14 p=91

0	1	1	1	1	1	1	0	0	0	0	1
1	1	0	0	0	0	0	0	1	1	1	1
0	1	1	0	1	1	1	1	1	1	0	1
1	1	1	0	1	1	0	0	0	1	0	0
1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	0	1	0	1	0	1	0	1
0	0	0	0	0	1	1	1	1	1	1	1
0	0	0	1	1	1	1	0	0	0	0	0
1	1	0	0	0	1	0	0	0	0	0	0
1	0	0	1	1	1	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	0	0	0
0	0	1	1	0	0	0	0	0	0	1	0
1	1	1	0	1	1	0	0	1	1	1	1
1	1	1	1	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	1	1	1	1	0
1	1	1	1	0	0	0	0	1	1	1	0
1	0	0	1	1	0	0	1	1	0	0	0
0	1	0	0	1	1	1	0	1	0	1	0
1	0	0	0	0	1	0	0	0	1	1	0
1	0	0	0	0	1	1	1	1	1	0	0
0	0	0	1	0	1	1	1	0	0	1	1
0	1	1	1	0	0	1	0	0	0	1	1
0	0	0	0	0	0	1	1	1	0	0	0
0	0	1	1	1	1	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	1	1	1
1	0	1	0	1	0	0	0	0	1	0	1
1	1	1	0	0	1	1	1	1	0	0	1
0	0	1	0	1	0	1	0	0	1	0	0
1	1	0	0	0	1	1	0	1	0	1	1
0	0	1	1	1	0	0	0	1	1	0	1
0	1	0	0	0	1	1	0	0	0	0	1
0	1	1	0	0	0	1	0	1	1	0	0
0	1	0	1	0	1	0	1	0	1	1	1
0	0	0	0	0	1	0	1	0	1	0	1
1	0	1	1	1	1	1	0	0	1	1	0

Figure 17: Deletion Code with n=12 s=2 m=36