FROM NATURAL LANGUAGE TO EXACT COVER: A NEURO-SYMBOLIC APPROACH TO ZEBRA PUZZLES

Paulius Skaisgiris[1]*, **Thomas Pammer**[2]*, **Veronika Semmelrock**[3], **Mykyta Ielanskyi**[4],
Maximilian Heisinger[5]† & **Erich Kobler**[4,6]†
[1]Institute of Logic and Computation, TU Wien
[2]Institute for Application-Oriented Knowledge Processing, JKU Linz
[3]Department of Artificial Intelligence and Cybersecurity, University of Klagenfurt
[4]Institute for Machine Learning, JKU Linz
[5]Institute for Symbolic Artificial Intelligence, JKU Linz
[6]LIT AI Lab, Department of Virtual Morphology, Clinical Research Institute Medical AI, JKU Linz
`paulius.skaisgiris@tuwien.ac.at`, `tpammer@faw.jku.at`,
`veronika.semmelrock@aau.at`, `ielanskyi@ml.jku.at`,
{`maximilian.heisinger`, `erich.kobler`}`@jku.at`

ABSTRACT

Chain-of-Thought (CoT) generation has substantially improved the performance of Large Language Models (LLMs) on complex reasoning tasks, including code generation, data analysis, and exam-style question answering. Despite these advances, purely neural LLMs continue to struggle with elementary logical reasoning problems and lack the determinism, soundness, and reliability characteristic of symbolic reasoning systems. Conversely, classical symbolic methods such as SAT solving and Exact Cover guarantee correctness and completeness, but require problems to be expressed in highly specialized formal encodings, limiting their applicability to natural language inputs. In this work, we present a tightly integrated neuro-symbolic framework that bridges this gap by combining neural semantic parsing with deterministic constraint solving. Our approach leverages the relational extraction capabilities of modern LLMs to parse Zebra-style logic puzzles written in free-form text and translate the extracted constraints into structured tool calls. These function calls assemble a formally specified Exact Cover instance, which is subsequently solved by a symbolic solver to ensure logically consistent solutions. We conduct a comprehensive empirical evaluation across multiple parameter scales, post-training paradigms, and LLM families. The results on larger puzzles demonstrate that our hybrid approach consistently outperforms strong plain neural baselines, including CoT prompting, as well as recent neuro-symbolic methods.[1]

1 INTRODUCTION

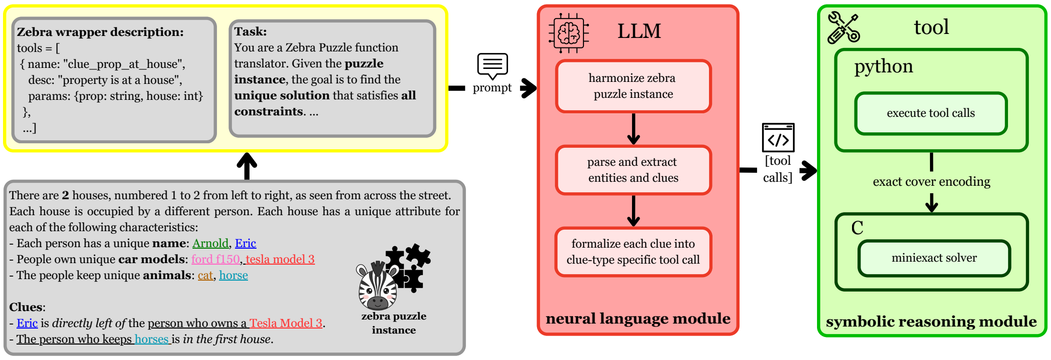
Constraint Satisfaction Problems (CSPs) lie at the core of many real-world decision-making tasks, including scheduling, planning, resource allocation, and configuration, making them a fundamental abstraction in AI research (Russell & Norvig, 1995; Mackworth, 1977; Nadel, 1989; Knuth, 2016; 2020; 2025). While symbolic methods such as SAT and constraint solvers can guarantee correctness and completeness, their practical deployment critically depends on constructing effective formal encodings – a process that remains difficult even for domain experts (Heisinger et al., 2024). Translating informal problem descriptions into a structured encoding requires substantial modeling expertise as small representational choices can dramatically impact solver performance and tractability. As a result, research often studies simplified yet representative proxy domains such as logic puzzles (e.g. Sudoku or Zebra Puzzles), which capture essential combinatorial and relational reasoning chal-

---

*Equal contribution

†Equal supervision

[1] Code: `https://codeberg.org/bilai-ss25-booleguru-llm/zebrallm`



Zebra wrapper description:

```
tools = [
 { name: "clue_prop_at_house",
   desc: "property is at a house",
   params: {prop: string, house: int}
 },
 ...]
```

Task:
You are a Zebra Puzzle function translator. Given the **puzzle instance**, the goal is to find the **unique solution** that satisfies **all constraints**. ...

There are **2** houses, numbered 1 to 2 from left to right, as seen from across the street. Each house is occupied by a different person. Each house has a unique attribute for each of the following characteristics:
- Each person has a unique **name**: Arnold, Eric
- People own unique **car models**: ford f150, tesla model 3
- The people keep unique **animals**: cat, horse

Clues:
- Eric is *directly left of* the person who owns a Tesla Model 3.
- The person who keeps horses is *in the first house*.

Figure 1: Visualization of our NEURALEXACTCOVER Zebra Puzzle solver. A puzzle instance is combined with our prompt template and fed into an LLM. First, the LLM harmonizes the puzzle to mitigate duplicate or ambiguous attributes. Second, all entities and clues are extracted to define Exact Cover items. Finally, each clue is transformed into exact cover options. The resulting encoding is solved using the symbolic MINIEXACT solver, computing the puzzle solution that is parsed into plain text.

lenges while providing clean semantics, controllable complexity, and objectively verifiable solutions (Dechter & Pearl, 1987; Prosser, 1993; Lin et al., 2025; White et al., 2025).

This reliance on structured benchmark domains aligns naturally with recent advances in Large Language Models (LLMs), which are increasingly applied to tasks requiring the construction of structured symbolic solutions from unstructured inputs. Such domains include mathematical reasoning (Hendrycks et al., 2021b), programming (Hendrycks et al., 2021a), planning (Zhang et al., 2026), and logic puzzle solving (Lin et al., 2025; Chollet et al., 2025). A shared characteristic of these problems is that, although discovering solutions is challenging, their correctness can be verified algorithmically. Reinforcement learning with verifiable rewards post-training has therefore emerged as a standard approach for optimizing models directly against symbolic verifiers (Guo et al., 2025). The resulting thinking LLMs produce Chain-of-Thought (CoT) reasoning traces enabling inference-time scaling.

Zebra Puzzles are among the recent benchmarks to test the reasoning capabilities of LLMs (Lin et al., 2025; White et al., 2025). These puzzles consist of lists of entities, properties, and attributes as well as clues; the goal is to infer the relations of attributes and entities from the given information. Lin et al. (2025) find that scaling the model size does not help beyond a certain level of complexity which is determined by the size of a puzzles' solution space. This manifests in generalist LLMs faring extremely well on simple Zebra Puzzles while struggling with more complex ones. Longer reasoning traces appear to be more helpful with these problems than larger models. However, these longer traces present additional challenges such as high frequency noise (Radler et al., 2025) and increased chance of hallucination (Mei et al., 2025).

To address the reasoning limitations of purely neural models, recent neuro-symbolic approaches increasingly combine LLM-based semantic parsing with symbolic solvers to enforce logical consistency (Dinu et al., 2024). Prior work explores both agentic formulations that iteratively generate solver-compatible programs and solver-centric pipelines that translate natural language inputs into formal constraint specifications solved using SAT, SMT, CSP, or ASP techniques (Szeider, 2025; Pan et al., 2023; Ye et al., 2023; Eiter et al., 2025). Notably, Logic.py (Kesseli et al., 2025) integrates LLM-based translation of natural language problems into a domain-specific language (Logic.py) representation that is subsequently solved using an exact constraint solver. This tight coupling of semantic parsing and symbolic solving consistently outperforms LLMs on Zebra-style logic benchmarks.

In this work, we present a tightly integrated neuro-symbolic framework called NEURALEXACTCOVER that combines LLM-based semantic parsing with deterministic Exact Cover solving for reliable logical reasoning over plain text Zebra-style puzzles. Our main contributions are as follows:

- We introduce and adapt an Exact Cover encoding for efficient symbolic solving of Zebra-style logic puzzles.
- We introduce a neuro-symbolic framework that leverages LLMs as a translation layer for an Exact Cover solver.
- We introduce the ZebraExtreme dataset, extending ZebraLogic (Lin et al., 2025) with even more challenging problems of size up to 8×10 entities and properties.

2 BACKGROUND AND PRELIMINARIES

2.1 EXACT COVER

Given a collection S of subsets of a set $\mathcal{X}$, the *exact cover problem* is about finding a subcollection $\mathcal{S}^* \subseteq \mathcal{S}$ such that each element in $\mathcal{X}$ is contained in *exactly one* subset of $\mathcal{S}^*$. This concept becomes clearer when we rename elements in $\mathcal{X}$ to *items* and $\mathcal{S}$ to *options*. We want to find the subcollection $\mathcal{S}^*$ of options to cover all items exactly once. One example comes from Knuth (2020): Let $\mathcal{X} = \{a, b, c, d, e, f, g\}$ be items and the collection of options $\mathcal{S} = \{\big[c, e\big], \big[a, d, g\big], \big[b, c, f\big], \big[a, d, f\big], \big[b, g\big], \big[d, e, g\big]\}$. When selecting the first, fourth, and fifth options, we cover each item *exactly* once, i.e. $\mathcal{S}^* = \{\big[c, e\big], \big[a, d, f\big], \big[b, g\big]\}$. This is very useful, as such exact-1 properties have a large variety of encodings with strong performance impacts for solvers, as discussed in the context of SAT-solving in Nguyen et al. (2020).

Colored Exact Cover. Exact cover can be extended by splitting the options into *primary items* $\mathcal{X}_P$ and *secondary items* $\mathcal{X}_S$ such that $\mathcal{X} = \mathcal{X}_P \cup \mathcal{X}_S$. Let $\mathcal{E}$ denote a set of colors that can be used to *color* (change) secondary items. Now, each primary item must still be covered *exactly once*, while secondary items must be assigned *at most one* color. Note that colors can be shared between options, i.e. multiple options with secondary items of matching colors may be selected.

Notation. We write options as sets of comma-separated items enclosed in brackets, e.g., $\big[a, b\big]$. We use colons to separate secondary items and colors. Thus, $a : c$ means the secondary item $a \in \mathcal{X}_S$ has the color $c \in \mathcal{E}$. Items may be combined from other terms using angle brackets, e.g. $\langle a, b \rangle$ is one item. For instance, $\big[\langle a, b \rangle : 1, c\big]$ denotes an option of one item $\langle a, b \rangle$ with the color 1 (meaning $\langle a, b \rangle$ must be a secondary item), and c as another item. A set of options is enclosed in curly brackets.

MINIEXACT is an efficient implementation[2] of some of the Dancing Links algorithms described by Knuth (2020). Its performance mostly mirrors Knuth's original C implementation, while also offering a user-friendly Python API and a more sophisticated parser. The algorithm can be selected at run-time, making it a useful tool for experiments with exact-cover problems.

2.2 ZEBRA PUZZLES

Zebra Puzzles are classic constraint satisfaction problems (CSPs) in which a solver must determine a one-to-one mapping between several categories of entities based on a set of logical clues, see Fig. 1 for an example.

Formulation. Each puzzle consists of initial statements defining a finite set of entities $\mathcal{E} = \{e_1, \ldots, e_N\}$ (e.g., numbered houses) with $N > 1$; several characteristics or properties $\mathcal{P} = \{p_1, \ldots, p_M\}$ (e.g., name, car, animal) with $M \geq 1$, each containing N mutually exclusive attributes; and a set of clues $\mathcal{C} = \{c_1, \ldots, c_K\}$ with $K \geq 1$ that relate entities, properties, and attributes. All attributes of a property define its domain $\mathcal{A}_p = \{a_{p1}, \ldots, a_{pN}\}$ (e.g., cat, horse) with $|\mathcal{A}_p| = N$.

Constraints. To ensure uniqueness of each puzzle's solution, *uniqueness* and *clue-based* constraints have to be fulfilled. The *uniqueness constraint* ensures that every attribute $a \in \mathcal{A}_p$ is assigned exactly to one entity for all $p \in \mathcal{P}$. *Clue-based constraints* formalize the different verbal clues into logic statements that must be satisfied by a valid solution. For Zebra Puzzles, there are 8

---

[2] `https://github.com/miniexact/miniexact`

different clue-based constraints that relate attributes and/or entities by specifying presence/absence or positional relations as summarized in Table 1.

Task. The goal is to find an assignment $x_{p,e} \in \mathcal{A}_p$ selecting an attribute for each property $p \in \mathcal{P}$ and entity $e \in \mathcal{E}$ such that all constraints are fulfilled. Note that these assignments can be summarized into a tabular grid where the columns are defined by the entities, the rows by the properties, and the assigned attributes are listed in the corresponding cells.

Datasets. Zebra Puzzles are used as proxies to test the reasoning capabilities of modern LLMs (Lin et al., 2025; White et al., 2025). To this end, Lin et al. (2025) introduced the ZebraLogic dataset containing 1,000 logic grid puzzles of various sizes ($N \times M$) ranging from 2×2 to 6×6. We extend this dataset to include 400 bigger puzzle instances (7×5, …, 8×10) and call this extended dataset ZebraExtreme. Following Lin et al. (2025), we partition the puzzles according to their solution space size ($|\mathcal{Q}| = (N!)^M$) into bins:

$$\begin{aligned} &\text{S}:\ 10^0 \leq |\mathcal{Q}| < 10^3, && \text{M}:\ 10^3 \leq |\mathcal{Q}| < 10^6, && \text{L}:\ 10^6 \leq |\mathcal{Q}| < 10^{10}, \\ &\text{XL}:\ 10^{10} \leq |\mathcal{Q}| < 10^{18}, && \text{XXL}:\ 10^{18} \leq |\mathcal{Q}| < 10^{30}, && \text{XXXL}:\ 10^{30} \leq |\mathcal{Q}| < 10^{50}. \end{aligned}$$

Note that the bins S to XL are included in the original ZebraLogic dataset, while the ZebraExtreme only includes puzzles in XXL and XXXL. Appx. A.2 describes how we created this computationally challenging dataset.

3 SOLVING ZEBRA PUZZLES WITH EXACT COVER

In the following, we present our NEURALEXACTCOVER approach for solving Zebra logic puzzles. As depicted in Figure 1, we leverage in-context learning capabilities of LLMs to extract semantic relations between entities, properties, and attributes from free-text Zebra Puzzles. The text description of the puzzles is first converted into an Exact Cover encoding, which is then used by the symbolic module to determine the assignment of attributes to entities. This way, the neural processing is done first, followed by the symbolic without any further neural postprocessing.

3.1 LANGUAGE MODULE FOR PUZZLE PARSING

The main goal of the language module is the conversion of the problem from natural language to a valid exact cover encoding. At the core of this module is a language model. To this end, we consider several recent open weight LLMs that are state-of-the-art for their parameter size, such as Qwen3 (Yang et al., 2025), gpt-oss (OpenAI et al., 2025), Magistral (Rastogi et al., 2025a), and Devstral (Rastogi et al., 2025b). Note that any language model that has been fine-tuned to perform tool calling could be used. The model is prompted to complete the following sequence of actions: *harmonization*, *clue extraction* and *formalization*. The full prompt is listed in Appx. D. It includes manually constructed few-shot examples to improve the adherence of the model to the specific instructions given.

Harmonization of the problem means extracting the properties in <property-name>_ <attribute-name> format. This is done to reduce the ambiguity of some attributes. For example, if “cat” appears as an attribute of two properties, this may lead to erroneous extraction of the attribute and subsequent incorrect encoding. The harmonized representation of the problem is still in natural language.

Once the natural text problem is harmonized, the LLM parses the puzzle’s setup (i.e. M, N, $\{\mathcal{A}_p\}_{p\in\mathcal{P}}$, and K) and proceeds to extract the clues. The clue extraction is intertwined with tool calls, as the model is urged to use the tools specified by the Exact Cover encoding presented in the next section. Consequently, the output of the LLM is a list of tool calls in JSON format, specifying the called function and its arguments. The first call must be to the initialization tool, that specifies the dimensions of the problem, i.e. number of entities, properties, property domains, and clues. The subsequent tool calls encode each clue. The complete reference of callable tools is provided in Appx. C.1. Note that we prompt each model to provide a one-sentence description for each tool call to simplify the optional inspection of traces (Appx. E).

Table 1: Exact cover encodings of all different clue types as sets of options. Here, $c \in \mathcal{C}$ denotes a clue, $a \in \mathcal{A}_p, \hat{a} \in \mathcal{A}_{\hat{p}}$ any attribute of a property $p, \hat{p} \in \mathcal{P}$, and $e, \hat{e} \in \mathcal{E}$ any entity. Note that we only list the exact cover encoding for the *left of* clues in the last lines; the *right of* pendants have similar structure but shifted indices.

Clue Type	**Exact Cover Encoding**
a at entity e	$\{[c,\ a : e]\}$
a not at entity e	$\bigcup_{\hat{e}\in\mathcal{E}}\{[c,\ a : \hat{e}]\}\setminus\{[c_\alpha,\ a : e]\}$
a at same entity as $\hat{a}$	$\bigcup_{\hat{e}\in\mathcal{E}}\{[c,\ a : \hat{e},\ \hat{a} : \hat{e}]\}$
a is Π entities from $\hat{a}$	$\bigcup_{i=1}^{N-\Pi}\{[c,\ a : e_i,\ \hat{a} : e_{i+\Pi}], [c,\ a : e_{i+\Pi},\ \hat{a} : e_i]\}$
a directly left/right of $\hat{a}$	$\bigcup_{i=1}^{N-1}\{[c,\ a : e_i,\ \hat{a} : e_{i+1}]\}$
a left/right of $\hat{a}$	$\bigcup_{i=1}^{N-1}\bigcup_{j=i}^{N}\{[c,\ a : e_i,\ \hat{a} : e_j]\}$

3.2 Reasoning Module using Exact Cover

We use Exact Cover as a logic backbone since it natively expresses exact-one constraints instead of requiring complex encodings as in SAT-solving (Nguyen et al., 2020).

Items. We assign a primary item to all clues $\mathcal{C}$, ensuring that every clue has to be used once. We expand entities and properties over their full domain and add a primary item for each pair, to model that each property has exactly one entity. Moreover, we use entities to colorize secondary items, defined by all attributes of all properties. Thus, the set of items is defined as

$$\mathcal{X} = \mathcal{X}_P \cup \mathcal{X}_S, \quad \mathcal{X}_P = \mathcal{C} \cup \left(\bigcup_{e\in\mathcal{E}} \bigcup_{p\in\mathcal{P}} \Big\{ [\langle p, e\rangle] \Big\} \right), \quad \mathcal{X}_S = \bigcup_{p\in\mathcal{P}} \bigcup_{a\in\mathcal{A}_p} \Big\{ [a] \Big\}.$$

Options. To model the uniqueness constraint – every attribute of each property can only be assigned to one entity, we set

$$\mathcal{S}_U = \bigcup_{e\in\mathcal{E}} \bigcup_{p\in\mathcal{P}} \bigcup_{a\in\mathcal{A}_p} \Big\{ [\langle p, e\rangle, a : e] \Big\}.$$

The clue-based constraints are transformed into corresponding exact cover encodings based on the 8 different clue types. Then, the set of options reads as

$$\mathcal{S} = \mathcal{S}_U \cup \mathcal{S}_\mathcal{C}, \quad \mathcal{S}_\mathcal{C} = \bigcup_{c\in\mathcal{C}} \mathcal{S}_c,$$

where S_c implements the encoding of clue c according to its clue type listed in Table 1. Because each clue has a primary item, exactly one of the clue's options must be in the solution. This forces the secondary items representing the affected attributes to be colored according to the clue, in turn affecting other clues and the solution as a whole.

Implementation. We use the MiniExact API to implement the encoding from above in Python. The whole encoding is contained in a Python class, which we instantiate for each new problem. To define the primary and secondary items and colors, we implement an `init_zebra` function with arguments: number of entities N, number of clues K, and a dictionary mapping each property to its list of attributes. To encode the clues, we implemented corresponding member functions for each clue type listed in Table 1. The functions include precise documentation and examples to facilitate in-context learning. We also add explicit synonym functions for abstract clue types to help LLMs select the correct type from textual clues. After processing all function calls, we run the MiniExact solver. If it does not converge within 120 seconds, we terminate it and return an empty solution. We provide further details along with the function descriptions in Appx. C.

While mapping clues to their corresponding types (encoding functions) is reliable, the exact attribute strings are not. LLMs sometimes use plural forms (e.g., *flowers* instead of *flower*), which creates

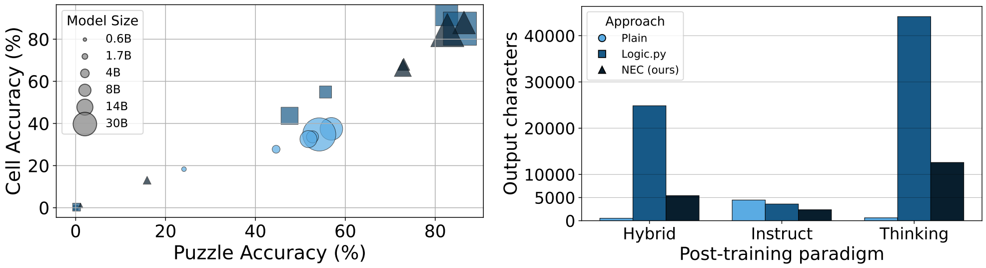


Figure 2: Average puzzle and cell accuracy on the ZebraLogic dataset for (left) varying Qwen 3 LLM family parameter scales and (right) average number of output characters by the LLM per post-training paradigm. Note that Logic.py generates more output characters (tokens) as it invokes the LLM multiple times on errors.

an issue since the evaluation requires exact matches of property and attribute names. Consequently, a puzzle may be encoded and solved consistently yet marked incorrect due to minor naming variations. To maintain naming uniformity, we use the initial puzzle setup function as the ground-truth reference. Thus, extracted names (arguments parsed by the LLM) are mapped to their canonical spelling from the puzzle setup using a string similarity ratio. The ratio is computed with the Ratcliff–Obershelp algorithm as implemented in Python's `difflib.SequenceMatcher`. This approach was more effective than prompt engineering or changing models, eliminating spelling mismatches that would otherwise confound our evaluation of LLM reasoning.

4 EXPERIMENTS

We evaluate the reasoning performance of plain LLMs (without tools), Logic.py (Kesseli et al., 2025), and our NEURALEXACTCOVER (NEC) on Zebra Puzzles. In particular, we consider the recent ZebraLogic benchmark (Lin et al., 2025) and our introduced ZebraExtreme, a more challenging variant characterized by a larger number of cells and an expanded set of possible values per cell. Refer to Appx A.2 for details on this dataset.

To assess the reasoning and scaling capabilities of the different methods, we evaluate puzzle and cell accuracy. While the former counts the fraction of correctly solved puzzles, the latter counts the fraction of correctly assigned attributes to entities (i.e. cells in the solution table). In addition, we evaluate the runtime of the LLMs and the symbolic solvers. We stratify the runtime and puzzle accuracy by the puzzle size into bins S to XXXL to analyze how models perform on increasingly complex puzzles.

For all methods, each LLM was run with `temperature` set to 0 and `top_p` set to 1, following Lin et al. (2025). We also adopt an output token budget of 4096 per LLM call, except for ZebraExtreme, where the budget is increased to 30720 to accommodate reasoning over larger puzzles. Notably, Logic.py Lin et al. (2025) employs a multi-step LLM pipeline that checks for syntax and constraint-encoding errors at each stage and reruns the LLM when such errors occur, thereby enabling multiple self-correction attempts. Consequently, the total token usage may exceed the output token budget of 4096, as the limit is enforced per individual call. In contrast, our approach relies on a single LLM call. Additional details on models and sampling parameters are provided in Appx A.1. As all three approaches under consideration are LLM-agnostic, we evaluate models spanning a diverse range of parameter scales, training paradigms, and model families.

Parameter Scales. In this experiment, we evaluate Qwen3 0.6B–30B parameter hybrid models (Yang et al., 2025), which support both thinking and non-thinking modes. This allows us to analyze how model scale impacts performance on Zebra Puzzles. The relationship between puzzle accuracy and cell accuracy across model sizes and approaches is shown on the left in Figure 2. Overall, smaller models (e.g., 0.6B and 1.7B) perform poorly, remaining well below 50% accuracy. As model size increases, performance improves, though gains diminish at larger scales, with results for 14B and 30B models lying relatively close together across approaches. The Plain approach

Table 2: Effect of training mode of Qwen3 LLM's. The best result per model is written in *italics*, while the overall best is highlighted in **bold**.

Model	Approach	Accuracies [%] ↑					
		All	S	M	L	XL	Cell
4B	Plain	52.7	*98.44*	71.07	6.5	0	33.65
4B	Logic.py	55.6	74.69	58.21	52	25	54.92
4B	NEC (ours)	73	93.75	81.07	65.5	36	68.08
4B-I	Plain	66.1	97.81	*85.71*	48	6	52.41
4B-I	Logic.py	*80.3*	91.88	*85.71*	*75*	*59.5*	*84.64*
4B-I	NEC (ours)	33.2	59.06	35.36	15	7	40.92
4B-T	Plain	27.5	79.06	7.86	0	0	13.54
4B-T	Logic.py	21	28.12	16.07	14	23.5	21.18
4B-T	NEC (ours)	1.2	3.44	0.36	0	0	0.47
30B	Plain	54.2	99.06	75	7.5	0	34.79
30B	Logic.py	85.5	93.44	87.5	82	73.5	85
30B	NEC (ours)	82.7	96.88	90	73.5	59	84.31
30B-I	Plain	70.9	***99.38***	***95.71***	55	6.5	54.28
30B-I	Logic.py	83.6	95.31	86.43	79.5	65	80.06
30B-I	NEC (ours)	***93.2***	98.44	95.36	***91***	***84***	***93.08***
30B-T	Plain	35.5	92.19	21.43	0	0	19.12
30B-T	Logic.py	71.6	73.75	76.79	71.5	61	72.46
30B-T	NEC (ours)	29.9	67.5	24.29	7.5	0	16.81

appears to plateau, reaching at most approximately 60% puzzle accuracy and 40% cell accuracy regardless of model size. Logic.py achieves competitive performance only with larger models (e.g., 14B and above). In contrast, NEURALEXACTCOVER performs well already at 4B and continues to outperform Logic.py at 4B and 8B, indicating a lower dependence on model scale. For full results, see Appx. Table 4.

LLM Post-Training Paradigms. Next, we examine the effect of instruction-following and thinking post-training paradigms on Zebra Puzzle performance. To this end, we evaluate the Qwen3 model family in hybrid (no suffix), instruct ("-I"), and thinking ("-T") configurations at both 4B and 30B parameter scales. Table 2 reports the corresponding puzzle and cell accuracies. Across all approaches, performance is high on small puzzles but degrades substantially as puzzle size increases. The Plain approach achieves near-perfect accuracy on small puzzles, but its performance drops sharply on larger instances, often failing entirely on XL puzzles. In contrast, Logic.py exhibits more stable performance across puzzle sizes, particularly for larger models, likely owing to its multi-step pipeline that enables intermediate error checking and repeated self-correction.

Our method outperforms both Plain and Logic.py in several settings without relying on self-correction, most notably for the 30B-I model, where it achieves the best overall performance across puzzle sizes and cell accuracy. However, its performance is less consistent at smaller scales, particularly for 4B-I and all thinking variants.

This behavior is further reflected in the output length statistics shown on the right in Figure 2. Logic.py produces substantially longer outputs, especially in hybrid and thinking configurations, consistent with its multi-step reasoning and self-correction pipeline. In contrast, both Plain and NEURALEXACTCOVER generate significantly shorter outputs, with NEURALEXACTCOVER remaining comparatively efficient across paradigms despite achieving strong performance.

Interestingly, the effect of post-training paradigm varies across approaches. For Plain, instruct models (e.g., 4B-I and 30B-I) outperform both hybrid and thinking variants, while thinking models perform substantially worse. For Logic.py and NEURALEXACTCOVER, a similar trend emerges, with instruct and hybrid variants generally outperforming thinking models. We found that, across all approaches, the thinking models underperform because their outputs are so long that they often fail to produce a final, evaluable solution, see Appx B.2 for a detailed evaluation. Moreover, while the 4B-I model performs poorly with NEC, its 30B counterpart leverages our approach more effectively than any other model, achieving the highest puzzle accuracy overall.

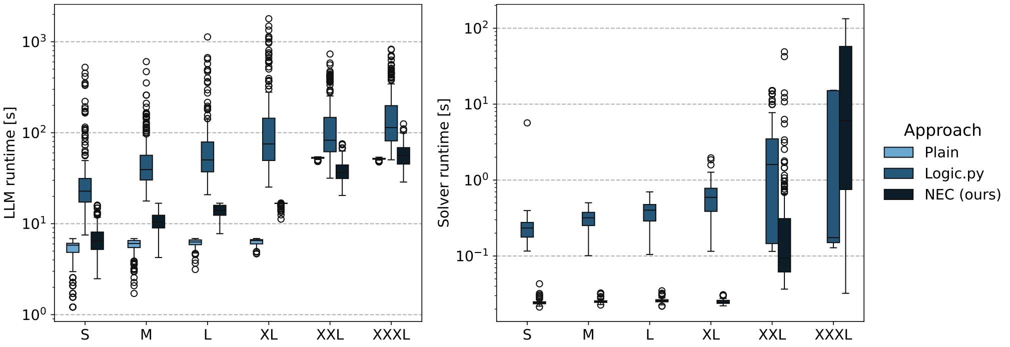


Figure 3: Runtimes of the LLM (left) and the symbolic solver (right) for gpt-oss-20b. Model output was capped at 4K tokens for S-XL and 30K for XXL-XXXL.

Performance across LLM families. We compare the performances of the three considered approaches with various LLM families in the 20B-30B range (details on models used can be found in Appx. A.1). The results are summarized in Tab. 3. Overall, NEURALEXACTCOVER performs strongly across most evaluated models, often achieving the highest puzzle and cell accuracies on ZebraLogic. In particular, it attains the best overall performance with Qwen3-30B-Instruct and Devstral Small-2, outperforming both Plain and Logic.py.

On the more challenging ZebraExtreme dataset, the Plain approach consistently fails to scale, yielding near-zero puzzle accuracy across all models, highlighting the limits of reasoning scalability for models without external tools. In contrast, both Logic.py and NEURALEXACTCOVER achieve non-trivial performance, with our method substantially outperforming Logic.py for several models. Notably, NEURALEXACTCOVER achieves the strongest ZebraExtreme results with gpt-oss (80% puzzle accuracy and 79.59% cell accuracy) and Qwen3-30B-Thinking, demonstrating improved scalability to larger problem instances.

At the time of writing, the official ZebraLogic state-of-the-art is 91.7% puzzle accuracy by o3-mini-2025-01-31-high[3] (Plain approach). Our NEURALEXACTCOVER with Qwen3-30B-A3B-Instruct achieves 93.2% puzzle accuracy, establishing a new SOTA. We also provide the first ZebraExtreme baseline, reaching 80% puzzle and 79.59% cell accuracy.

Runtime Comparison. We compare the runtime of all approaches for both generation and the symbolic solver on the ZebraLogic and ZebraExtreme datasets (see Fig. 3). Here, we focus on the gpt-oss-20b model. For Logic.py and NEURALEXACTCOVER, LLM runtimes increase roughly an order of magnitude from S to XXXL, reflecting the growing problem complexity, while our method remains consistently faster by approximately half an order or magnitude across all sizes. In contrast, the Plain approach exhibits roughly constant median times up to size XL and hits the maximum number of output tokens beyond.

Regarding solver performance, NEURALEXACTCOVER outpaces Logic.py by roughly one order of magnitude on S–XL ZebraLogic puzzles, exhibiting very fast symbolic solver times. On ZebraExtreme, this advantage is maintained in the median performance for XXL and disappears at XXXL. Despite similar runtime at the largest XXXL scale, NEURALEXACTCOVER achieves a substantial accuracy improvement over Logic.py, with a puzzle accuracy gain of 56.42% on XXL and 55.88% on XXXL (see Tab. 3).

5 RELATED WORK

Recent work addressing the reasoning limitations of LLMs increasingly combines neural models with symbolic solvers to enforce logical consistency. Among these approaches Logic.py (Kesseli et al., 2025), provides the closest conceptual and empirical point of comparison, integrating semantic

---

[3] Leaderboard accessed 2026-03-03 at `https://huggingface.co/spaces/allenai/ZebraLogic`.

Table 3: Performance by model family. The overall best performance per column is marked **bold**, while the best performance per LLM family is in *italics*. Model output was capped at 4K tokens for ZebraLogic and 30K for ZebraExtreme.

Model			ZebraLogic						ZebraExtreme			
	Size	Approach	All	S	M	L	XL	Cell	All	XXL	XXXL	Cell
Qwen 3-I	30B	Plain	70.9	***99.38***	***95.71***	55	6.5	54.28	0.5	0.83	0	14.58
	30B	Logic.py	83.6	95.31	86.43	79.5	65	80.06	4.5	9	0	7.25
	30B	NEC (ours)	***93.2***	98.44	95.36	***91***	*84*	*93.08*	*38*	*43.33*	*30*	*52.72*
Qwen 3-T	30B	Plain	35.5	*92.19*	21.43	0	0	19.12	1.25	2.08	0	1.81
	30B	Logic.py	*71.6*	73.75	*76.79*	*71.5*	*61*	*72.46*	8	15	1	11.9
	30B	NEC (ours)	29.9	67.5	24.29	7.5	0	16.81	*55.75*	*68.75*	*36.25*	*63.35*
Devstral Small-2	24B	Plain[†]	–	–	–	–	–	–	–	–	–	–
	24B-I	Logic.py	88.5	95.31	91.43	*89.5*	72.5	84.15	5	10	0	7.3
	24B-I	NEC (ours)	*93.1*	*97.5*	*93.93*	*89.5*	***88.5***	***94.63***	*49*	*56.25*	*38.12*	*53.04*
Magistral Small	24B	Plain	57.2	94.38	74.29	30	1	40.96	0	0	0	7.64
	24B	Logic.py	*89.4*	*95*	*91.79*	*88*	*78.5*	*87.14*	10	19	1	14.73
	24B	NEC (ours)	81	91.88	85.71	75.5	62.5	82.62	*26*	*30.42*	*19.38*	*35.79*
gpt oss	20B	Plain	48.9	90.94	61.07	13.5	0	35.83	0 [⋆]	0 [⋆]	0 [⋆]	0 [⋆]
	20B	Logic.py	*84.4*	95.62	88.21	*79*	*66.5*	*81.62*	21.25	34	8.5	23.64
	20B	NEC (ours)	72.7	*98.75*	*91.07*	66.5	11.5	56.66	***80***	***90.42***	***64.38***	***79.59***

[†] A template error prevented this configuration from returning any output.

[⋆] All LLM traces reached output token budget before returning any solution.

parsing with exact constraint solving. CP-Agent (Szeider, 2025) explores a fully agentic architecture in which a Python coding agent, guided by layered prompts, iteratively models and solves constraint problems, translated from natural language. In contrast to DSL-based pipelines, the system emphasizes iterative code generation and execution within a persistent runtime environment. Berman et al. (2024) propose a multi-agent framework for Zebra logic puzzles, where LLMs iteratively generate and refine SMT-LIB encodings that are solved by an external theorem prover. Solver feedback is incorporated into a refinement loop to correct inconsistencies. LOGIC-LM (Pan et al., 2023) and SATLM (Ye et al., 2023) translate natural language inputs into declarative symbolic specifications that are solved using logic programming systems, CSP, SAT, or SMT solvers. LOGIC-LM further incorporates optional solver feedback to refine symbolic encodings, while SATLM fully delegates planning and execution to the solver, thereby mitigating common reasoning errors associated with chain-of-thought prompting and providing correctness guarantees with respect to the parsed specification. Allen et al. (2025) integrate LLM outputs directly into a logical reasoning framework, treating the model as a source of truth judgments while keeping the reasoning itself symbolically formal, aiming to maintain logical guarantees even when LLM outputs are inconsistent. Chen (2025) compares integrative approaches, where reasoning is built into neural networks, with hybrid systems that translate problems into symbolic form and solve them externally, leading to hybrid architectures, which are generally more scalable and easier to interpret. Eiter et al. (2025) extend hybrid reasoning to the multimodal domain, integrating LLM-based semantic parsing with Answer Set Programming for explainable zero-shot visual question answering, highlighting the modularity and traceability of solver-based designs.

6 CONCLUSION

In this work, we introduce NEURALEXACTCOVER, a neuro-symbolic approach combining modern LLMs with an Exact Cover solver to solve Zebra-style puzzles. To the best of our knowledge, this is the first approach to provide LLMs with this type of logic solver as a callable tool. On ZebraLogic, our method matches recent approaches in accuracy while being more runtime-efficient, and it establishes a new state of the art with 93.2% puzzle accuracy. We introduce ZebraExtreme, a dataset featuring substantially larger puzzles than ZebraLogic. With a larger output token budget, previous approaches solve only a few of these challenging puzzles ($< 21.25\%$), whereas our method achieves 80% puzzle accuracy.

ACKNOWLEDGMENTS

The authors acknowledge support from the DFG within the SPP2298 under project number 543939932 and from the Austrian Science Fund (FWF) project number 10.55776/COE12.

REFERENCES

Bradley P. Allen, Prateek Chhikara, Thomas Macaulay Ferguson, Filip Ilievski, and Paul Groth. Sound and complete neurosymbolic reasoning with llm-grounded interpretations. *arxiv 2507.09751*, 2025. doi: 10.48550/ARXIV.2507.09751.

Shmuel Berman, Kathleen R. McKeown, and Baishakhi Ray. Solving zebra puzzles using constraint-guided multi-agent systems. *arxiv 2407.03956*, 2024. doi: 10.48550/ARXIV.2407.03956.

Michael K. Chen. A comparative study of neurosymbolic AI approaches to interpretable logical reasoning. *arxiv 2508.03366*, 2025. doi: 10.48550/ARXIV.2508.03366.

François Chollet, Mike Knoop, Gregory Kamradt, Bryan Landers, and Henry Pinkard. ARC-AGI-2: A new challenge for frontier AI reasoning systems. *arxiv 2505.11831*, 2025. doi: 10.48550/ARXIV.2505.11831.

Rina Dechter and Judea Pearl. Network-based heuristics for constraint-satisfaction problems. *Artif. Intell.*, 34(1):1–38, 1987. doi: 10.1016/0004-3702(87)90002-6.

Marius-Constantin Dinu, Claudiu Leoveanu-Condrei, Markus Holzleitner, Werner Zellinger, and Sepp Hochreiter. Symbolicai: A framework for logic-based approaches combining generative models and solvers. In *Conference on Lifelong Learning Agents*, volume 274 of *Proceedings of Machine Learning Research*, pp. 869–914, 2024.

Thomas Eiter, Jan Hadl, Nelson Higuera Ruiz, Lukas Lange, Johannes Oetsch, Bileam Scheuvens, and Jannik Strötgen. Explainable zero-shot visual question answering via logic-based reasoning. In *International Conference on Neurosymbolic Learning and Reasoning*, volume 284 of *Proceedings of Machine Learning Research*, pp. 977–991, 2025.

Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Peiyi Wang, Qihao Zhu, Runxin Xu, Ruoyu Zhang, Shirong Ma, Xiao Bi, and et al. Deepseek-r1 incentivizes reasoning in llms through reinforcement learning. *Nat.*, 645(8081):633–638, 2025. doi: 10.1038/S41586-025-09422-Z.

Maximilian Heisinger, Simone Heisinger, and Martina Seidl. Booleguru, the propositional polyglot (short paper). In *Automated Reasoning - International Joint Conference (IJCAR)*, pp. 315–324. Springer, 2024. doi: 10.1007/978-3-031-63498-7_19.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. Measuring Coding Challenge Competence With APPS. In *Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021a.

Dan Hendrycks, Collin Burns, Saurav Kadavath, Akul Arora, Steven Basart, Eric Tang, Dawn Song, and Jacob Steinhardt. Measuring Mathematical Problem Solving With the MATH Dataset. In *Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1, 2021b.

Pascal Kesseli, Peter W. O'Hearn, and Ricardo Silveira Cabral. Logic.py: Bridging the gap between llms and constraint solvers. *arxiv 2502.15776*, 2025. doi: 10.48550/ARXIV.2502.15776.

Donald Knuth. *The Art of Computer Programming. Volume 4, Fascicle 6. Satisfiability*. Addison-Wesley, 2016.

Donald Knuth. *The Art of Computer Programming. Volume 4, Fascicle 5. Mathematical Preliminaries Redux; Introduction to Backtracking; Dancing Links*. Addison-Wesley, 2020.

Donald Knuth. *The Art of Computer Programming. Volume 4, Fascicle 7. Constraint Satisfaction*. Addison-Wesley, 2025.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Symposium on Operating Systems Principles*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165.

Bill Yuchen Lin, Ronan Le Bras, Kyle Richardson, Ashish Sabharwal, Radha Poovendran, Peter Clark, and Yejin Choi. Zebralogic: On the scaling limits of llms for logical reasoning. In *International Conference on Machine Learning*, volume 267 of *Proceedings of Machine Learning Research*, 2025.

Alan K. Mackworth. Consistency in networks of relations. *Artif. Intell.*, 8(1):99–118, 1977. doi: 10.1016/0004-3702(77)90007-8.

Zhiting Mei, Christina Zhang, Tenny Yin, Justin Lidard, Ola Shorinwa, and Anirudha Majumdar. Reasoning about uncertainty: Do reasoning models know when they don't know? *arxiv 2506.18183*, 2025. doi: 10.48550/ARXIV.2506.18183.

Bernard A. Nadel. Constraint satisfaction algorithms. *Comput. Intell.*, 5:188–224, 1989. doi: 10.1111/J.1467-8640.1989.TB00328.X.

Van-Hau Nguyen, Van-Quyet Nguyen, Kyungbaek Kim, and Pedro Barahona. Empirical study on sat-encodings of the at-most-one constraint. In *International Conference on Smart Media and Applications*, pp. 470–475. ACM, 2020. doi: 10.1145/3426020.3426170.

OpenAI, Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K. Arora, Yu Bai, Bowen Baker, and et al. Gpt-oss-120b & gpt-oss-20b Model Card. *arXiv 2508.10925*, 2025.

Liangming Pan, Alon Albalak, Xinyi Wang, and William Yang Wang. Logic-lm: Empowering large language models with symbolic solvers for faithful logical reasoning. In *Findings of the Association for Computational Linguistics: EMNLP*, pp. 3806–3824, 2023. doi: 10.18653/V1/2023.FINDINGS-EMNLP.248.

Patrick Prosser. Hybrid algorithms for the constraint satisfaction problem. *Comput. Intell.*, 9:268–299, 1993. doi: 10.1111/J.1467-8640.1993.TB00310.X.

Andreas Radler, Vincent Seyfried, Stefan Pirker, Johannes Brandstetter, and Thomas Lichtenegger. PAINT: parallel-in-time neural twins for dynamical system reconstruction. *arxiv 2510.16004*, 2025. doi: 10.48550/ARXIV.2510.16004.

Abhinav Rastogi, Albert Q Jiang, Andy Lo, Gabrielle Berrada, Guillaume Lample, Jason Rute, Joep Barmentlo, Karmesh Yadav, Kartik Khandelwal, and et al. Magistral. *arXiv 2506.10910*, 2025a.

Abhinav Rastogi, Adam Yang, Albert Q Jiang, Alexander H Liu, Alexandre Sablayrolles, Amélie Héliou, Amélie Martin, Anmol Agarwal, Andy Ehrenberg, Andy Lo, et al. Devstral: fine-tuning language models for coding agent applications. *arXiv 2509.25193*, 2025b.

Stuart J. Russell and Peter Norvig. *Artificial intelligence - a modern approach: the intelligent agent book*. Prentice Hall series in artificial intelligence. Prentice Hall, 1995. ISBN 978-0-13-103805-9.

Stefan Szeider. Cp-agent: Agentic constraint programming. *arxiv 2508.07468*, 2025. doi: 10.48550/ARXIV.2508.07468.

Colin White, Samuel Dooley, Manley Roberts, Arka Pal, Benjamin Feuer, Siddhartha Jain, Ravid Shwartz-Ziv, Neel Jain, Khalid Saifullah, Sreemanti Dey, and et al. Livebench: A challenging, contamination-limited LLM benchmark. In *International Conference on Learning Representations*, 2025.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, and et al. Qwen3 Technical Report. *arXiv 2505.09388*, 2025.

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. Satlm: Satisfiability-aided language models using declarative prompting. In *Neural Information Processing Systems*, 2023.

Yinger Zhang, Shutong Jiang, Renhao Li, Jianhong Tu, Yang Su, Lianghao Deng, Xudong Guo, Chenxu Lv, and Junyang Lin. Deepplanning: Benchmarking long-horizon agentic planning with verifiable constraints. *arXiv 2601.18137*, 2026. doi: 10.48550/ARXIV.2601.18137.

A DETAILS ON DATASETS AND MODELS USED.

A.1 HARDWARE DETAILS, MODEL REFERENCE, AND SAMPLING PARAMETERS

The evaluation was carried out by serving each LLM on a server using vLLM (Kwon et al., 2023) on a single NVIDIA RTX PRO 6000 GPU with 96GB vRAM. The server node has AMD EPYC 9555 64-Core Processor and 2.2 TiB of RAM.

LLM weights were downloaded from Hugging Face. The exact Hugging Face identifiers of models used in this paper are as follows:

- `Qwen/Qwen3-0.6B`
- `Qwen/Qwen3-1.7B`
- `Qwen/Qwen3-4B`
- `Qwen/Qwen3-8B`
- `Qwen/Qwen3-14B`
- `Qwen/Qwen3-30B-A3B`
- `Qwen/Qwen3-4B-Instruct-2507`
- `Qwen/Qwen3-4B-Thinking-2507`
- `Qwen/Qwen3-30B-A3B-Instruct-2507`
- `Qwen/Qwen3-30B-A3B-Thinking-2507`
- `mistralai/Devstral-Small-2-24B-Instruct-2512`
- `mistralai/Magistral-Small-2506`
- `openai/gpt-oss-20b`

Unless otherwise stated, we used the default tokenizer and configuration provided with each repository. All models were executed without additional fine-tuning or weight modification.

All models were run with `temperature` set to 0, `top_p` to 1.0, and `max_model_len` to 30720, and were served on a single shard on one GPU. Evaluation for ZebraLogic used a 4096-token budget per LLM call, following Lin et al. (2025). For ZebraExtreme, the maximum output length was 30K tokens (the `max_model_len` limit), or shorter if the model stopped earlier. For the Plain approach, the tokenizer mode was set to "auto" for all models, except Magistral Small and Devstral Small, for which it was set to "magistral". For Logic.py and NEC, each model was served with `--enable-auto-tool-choice` flag and `--tool-call-parser` set to the recommended option for each LLM family, specified in the vLLM tool-calling documentation[4].

A.2 GENERATING ZEBRAEXTREME

The ZebraExtreme dataset comprises two size classes, XXL and XXXL, defined by their solution space size $|\mathcal{Q}| = (N!)^M$ representing the number of possible table fillings (with repetition). We designed the dataset with varied dimensions to investigate how increasing entities (N) or properties (M) impacts puzzle difficulty. The classes are assigned as follows:

---

[4] `https://docs.vllm.ai/en/latest/features/tool_calling/`

XXL		XXXL	
Size	$\lvert\mathcal{Q}\rvert$	Size	$\lvert\mathcal{Q}\rvert$
7×5	3.25×10^{18}	7×9	2.10×10^{33}
7×6	1.64×10^{22}	8×7	1.73×10^{32}
7×7	8.26×10^{25}	8×8	6.98×10^{36}
7×8	4.16×10^{29}	8×9	2.82×10^{41}
8×6	4.30×10^{27}	8×10	1.14×10^{46}

The pseudocode for our generator follows the high-level recipe from the Hugging Face article `https://huggingface.co/blog/yuchenlin/zebra-logic` (Puzzle Creation section):

1. Choose N houses and M features; each feature has N distinct values.
2. Sample a hidden solution (a Latin rectangle-like one-to-one assignment).
3. Enumerate candidate clues from a small set of clue types.
4. Iteratively remove clues while ensuring the remaining clues still imply a *unique* solution equal to the hidden one.

A.3 Logic.py setup adaptions

We set the temperature to 0 and top-p to 1 so that we have the identical settings in all of our three approaches. To ensure a fair comparison between NeuralExactCover and the Logic.py approach across different LLM back-ends, we extended the original system prompt of Logic.py as follows:

```
**Important modeling requirements for this task (do not ignore):**
- Use a **collection** in the top-level solution: `houses: list[House,
↪  N]` (here N=2).
- Each `House` must contain a numeric **position** field, e.g. `number:
↪  Unique[Domain[int, range(1, 3)]]`.
- Do not define separate fields like `house\_1` or `house\_2`.

**Output format requirement:**
Return **only one** fenced Python code block with the class definitions;
↪  no prose before or after.

Return only a single fenced Python code block. Do not include any
↪  analysis, explanations, or think content
```

The remainder of the workflow remained unchanged. We retained a retry limit of five attempts for unsolved puzzles and did not impose an explicit restriction on the number of generated output tokens. Since we deployed the model using a vLLM serve command configured with a maximum output length of 30K tokens, the algorithm was effectively allowed to produce longer responses. However, because Logic.py issues separate LLM calls for each step, empirical tests indicate that this increased token limit does not significantly affect the overall results.

B Additional Results

B.1 Evaluation for varying Parameter Scales

Table 4 shows the performance of the different approaches on different parameter scales of the Qwen3 models. As we can see, our NeuralExactCover already achieves $> 70\%$ puzzle accuracy with a 4B model, showing its robustness on the model scale. While the Plain baseline performs strongly on small puzzles, its accuracy decreases for larger instances. In contrast, NeuralExactCover maintains stable performance as puzzle size increases, resulting in higher cell accuracies overall. The performance drop from 14B to 30B is expected, since the 30B model uses only 3B active parameters, whereas the 14B model is dense, with all 14B parameters active. Note that Logic.py uses a multi-step LLM pipeline and can rerun calls when the output syntax is invalid. Consequently, although each call was limited to 4K output tokens, the total output token usage was much higher.

Table 4: Performance across Qwen3-base model sizes. The overall best performance per column is marked **bold**, while the best performance per size is written in *italics*.

Model		Accuracies [%] ↑					
Size	Approach	All	S	M	L	XL	Cell
0.6B	Plain	*24.1*	*61.25*	*15*	*1.5*	*0*	*18.31*
0.6B	Logic.py	0	0	0	0	*0*	0
0.6B	NEC (ours)	1.1	3.44	0	0	*0*	1.05
1.7B	Plain	*44.6*	*92.81*	*48.93*	*6*	0	*27.76*
1.7B	Logic.py	0.2	0.62	0	0	0	0.28
1.7B	NEC (ours)	15.9	37.19	11.07	4	*0.5*	13.04
4B	Plain	52.7	*98.44*	71.07	6.5	0	33.65
4B	Logic.py	55.6	74.69	58.21	52	25	54.92
4B	NEC (ours)	*73*	93.75	*81.07*	*65.5*	*36*	*68.08*
8B	Plain	51.8	***99.06***	67.14	6.5	0	32.57
8B	Logic.py	47.6	69.06	40.71	38	*32.5*	43.64
8B	NEC (ours)	*72.8*	97.81	*86.07*	*63.5*	23.5	*66.36*
14B	Plain	56.9	***99.06***	81.07	12.5	0	37.44
14B	Logic.py	82.5	72.19	82.5	***92.5***	***89***	***91.79***
14B	NEC (ours)	***86.4***	96.56	*87.86*	84	70.5	87.92
30B	Plain	54.2	***99.06***	75	7.5	0	34.79
30B	Logic.py	*85.5*	93.44	87.5	*82*	*73.5*	*85*
30B	NEC (ours)	82.7	96.88	***90***	73.5	59	84.31

We believe NeuralExactCover could similarly benefit from a self-correcting feedback loop, which we leave for future work.

B.2 Evaluation for 30K output token budget

After the poor performance of thinking models with a 4K output token budget in Tab. 2, we evaluated a larger 30K budget. Tab. 5 compares 4K vs. 30K for all models. Logic.py's results are unchanged, as its effective budget was already larger due to multiple LLM calls. Notably, increasing the budget by a factor of 7.5 yields only modest gains in puzzle and cell accuracy for most models, with substantial improvements occurring mainly for reasoning models. This is most likely because the hybrid and instruct models can output a candidate solution with a much smaller token budget. The biggest improvements can be observed for Qwen3-4B-Thinking (from 1.2% to 95.1% puzzle accuracy), Qwen3-30B-Thinking (from 29.9% to 96.5% puzzle accuracy) and gpt-oss-20b (from 72.7% to 96.7%). This confirms our hypothesis that the original output token budget was too small for thinking models. Notably, with NeuralExactCover, a larger output budget, and the relatively small Qwen3-4B-Thinking model, we can already nearly solve the ZebraLogic dataset.

C Implementation Details for MiniExact

MiniExact offers two modes of interaction: The command-line tool MiniExact and the API, either in the *simple* variant also exposed to Python, or the *direct* variant that the MiniExact binary is using. Both modes provide options to choose the solving algorithm to be used. This work relies on Algorithm C (Knuth (2020)) to solve Exact Cover problems with colored secondary items. The simple API requires consumers must to adhere to the following steps when using the provided `Solver` object:

1. Declare all primary items using `.primary`.
2. Declare all secondary items using `.secondary`.
3. Add all options using `.add`. We wrapped this in an `add_option` function that automatically ends each option.

Table 5: Overview of puzzle and cell accuracies for output token limit of 4K and 30K evaluated on ZebraLogic dataset. Best within each model is written in *italics*, the best across all models is ***bold italicised***.

Model	Mode	Output token limit 4K		Output token limit 30K	
		Puzzle acc ↑	Cell acc ↑	Puzzle acc ↑	Cell acc ↑
Qwen3-0.6B	Plain	*24.1*	*18.31*	*28.4*	*41.07*
	Logic.py	0	0	0	0
	NEC (ours)	1.1	1.05	1.3	1.31
Qwen3-1.7B	Plain	*44.6*	*27.76*	*62.8*	*69.33*
	Logic.py	0.2	0.28	0.2	0.28
	NEC (ours)	15.9	13.04	22	24.04
Qwen3-4B	Plain	52.7	33.65	*81.9*	*83.29*
	Logic.py	55.6	54.92	55.6	54.92
	NEC (ours)	*73*	*68.08*	78	78.96
Qwen3-4B-Instruct	Plain	66.1	52.41	79.3	82.42
	Logic.py	*80.3*	*84.64*	*80.3*	*84.64*
	NEC (ours)	33.2	40.92	32.8	40.37
Qwen3-4B-Thinking	Plain	*27.5*	13.54	92	92.82
	Logic.py	21	*21.18*	21	21.18
	NEC (ours)	1.2	0.47	*95.1*	*94.93*
Qwen3-8B	Plain	51.8	32.57	*84.1*	*84*
	Logic.py	47.6	43.64	47.6	43.64
	NEC (ours)	*72.8*	*66.36*	80.3	81.27
Qwen3-14B	Plain	56.9	37.44	*89.2*	88.76
	Logic.py	82.5	*91.79*	82.5	*91.79*
	NEC (ours)	*86.4*	87.92	87.1	89.67
Qwen3-30B	Plain	54.2	34.79	*88.7*	*88.62*
	Logic.py	*85.5*	*85*	85.5	85
	NEC (ours)	82.7	84.31	84.3	87.36
Qwen3-30B-Instruct	Plain	70.9	54.28	88.7	87.72
	Logic.py	83.6	80.06	83.6	80.06
	NEC (ours)	***93.2***	*93.08*	*92.7*	*93.06*
Qwen3-30B-Thinking	Plain	35.5	19.12	96	96.14
	Logic.py	*71.6*	*72.46*	71.6	72.46
	NEC (ours)	29.9	16.81	*96.5*	***96.49***
Devstral-Small-2-24B-Instruct	Plain[†]	–	–	–	–
	Logic.py	88.5	84.15	88.5	84.15
	NEC (ours)	*93.1*	***94.63***	*94.7*	*95.67*
Magistral-Small-24B	Plain	57.2	40.96	76.9	74.65
	Logic.py	*89.4*	*87.14*	*89.4*	*87.14*
	NEC (ours)	81	82.62	81	82.58
gpt-oss-20b	Plain	48.9	35.83	64.9	57.11
	Logic.py	*84.4*	*81.62*	84.4	81.62
	NEC (ours)	72.7	56.66	***96.7***	*96.4*

[†] A template error prevented this configuration from returning any output.

4. Call `.solve`, returning either 10 if there is a solution, or 20 if no (more) solution was found found. Multiple solutions may be extracted, but in this work, we only use the first one.

5. We extract the solution by calling `.print_solution` and parse the resulting string, as this was easier compared to keeping track of the indices to the inserted colors.

C.1 PYTHON API FOR EXACT COVER ENCODING

Encoding Function	Description
`init_zebra`	Initializes the puzzle with N entities M property groups, and K clues.
`clue_property_left_of`	Attribute a is somewhere to the left of attribute $\hat{a}$.
`clue_property_right_of`	Attribute a is somewhere to the right of attribute $\hat{a}$.
`clue_property_at_house`	Attribute a is fixed at entity e.
`clue_property_at_property`	Attributes a and $\hat{a}$ occur at the same house.
`clue_not_property_at_house`	Attribute a is not located at entity e.
`clue_property_next_to_property`	Attributes a and $\hat{a}$ are at adjacent entities.
`clue_one_between_two_properties`	Exactly one entity lies between attributes a and $\hat{a}$.
`clue_two_between_two_properties`	Exactly two houses lie between attributes a and $\hat{a}$.
`clue_property_directly_left_of_property`	Attribute a is immediately to the left of attribute $\hat{a}$.
`clue_property_directly_right_of_property`	Attribute a is immediately to the right of attribute $\hat{a}$.

D SYSTEM PROMPT USED FOR THE LANGUAGE MODULE

We provide the full system prompt used to instruct the language model to translate Zebra logic puzzles into structured tool calls. The prompt first defines the model role and expected input format.

```
You are a Zebra Puzzle function translator. Your role is to convert the
textual description of a Zebra Puzzle (a logic grid puzzle) into a
↪  sequence
of structured function calls that can programmatically solve it.

### Input Format
- The puzzle description begins with `=== PUZZLE ===`, including details
like the number of houses/places, and lists of property groups
↪  (categories
with unique values for each house).
- The clues begin with `### CLUES`, numbered from 1 to N.
```

The next section specifies the ordered transformation procedure applied by the model, including harmonization, initialization, and clue translation.

```
### Task Sequence
1. **Harmonize**: Re-print the input, but replace each property-value in
the puzzle *and* clues by <property-name>_<property-value>. Properties
should always be combined with their names. All spaces in properties
↪  should
be replaced by minus (-) symbols. All property names must be consistent
with the initial definition. Try to map similar words to
<property-name>_<property-value> tuples, e.g. something ending with
-ian or -ist is probably linked to its base or similar value, like
Scientist -> science, Austrian -> austria.
Be careful to convert each property in each clue to the fitting
↪  name_value
tuple. Use as many hints from the clue as possible, e.g. if the clue
↪  lists
a person's child's name to be Max, the property will be child_max.
2. **Initialize**: Use your harmonized puzzle. Extract the number of
houses/places, the property groups (each group is a category with its
unique values), and the total number of clues.
```

```
Output exactly one call to `init_zebra`. Use the same
<property-name>_<property-value> format for properties.
3. **Translate Clues**: For each numbered clue (1 to N) in your
 ↪  harmonized
puzzle, convert it into exactly one function call that represents the
 ↪  logical
constraint (e.g., a property in a specific house, a negation, adjacency,
etc.). Use appropriate function names based on the clue type (examples:
`clue_property_at_house`, `clue_not_property_at_house`,
`clue_property_next_to_property`).
Use the same <property-name>_<property-value> format for properties.
4. **Explain**: For each function call (including initialization),
 ↪  provide
a single-sentence explanation immediately before it, describing why the
parameters match the puzzle element or clue.
```

We then define strict output formatting and normalization rules that ensure consistent structured tool invocation.

```
### Output Format Rules
- Wrap every function call in `<tool_call>...</tool_call>` tags.
- Property and value names must be lowercase. Spaces are converted to -.
Property names are prepended, separated with _. Multiple words must
 ↪  remain
unshortened and in full, following the required format (e.g., "Labrador
Retriever" → "pet_labrador-retriever"; "Golden Gate Bridge" →
"location_golden-gate-bridge").
- Output exactly 1 initialization call followed by exactly N clue calls
(where N is the number of clues). No additional text or calls, beside the
harmonized output. Always be consistent with naming properties.
- Use the exact JSON-like structure in the template below for each call.
- Process the puzzle mechanically and systematically: Parse elements
 ↪  directly
without adding interpretations or solving the puzzle yourself. Only use
your harmonized puzzle for translating.
- Never correct yourself. You can directly translate clues individually,
without referring to other clues. New information is combined by the tool
afterwards.
```

The prompt also provides a template illustrating the required tool-call structure.

~~~
### Template
```
Harmonized puzzle:
...

Explanation: [One sentence explaining why this function and its
 ↪  parameters
represent the clue or initialization.]
<tool_call>
{
  "name": "function_name",
  "arguments": {
    "param1": "value1",
    "param2": "value2",
    ...
  }
}
</tool_call>
```
~~~

Additional processing guidelines constrain normalization and mapping behavior.

```
### Processing Guidelines
- Identify property groups from the puzzle description (e.g., categories
like "pet" with values ["pet_cat", "pet_dog"]).
```

````
- For clues, map to the simplest matching function (e.g., "X is in house
Y" → `clue_property_at_house` with `prop: "x"` and `house: Y`).
- Ensure all values and names are normalized (lowercase, minus (-)
↪  instead
of spaces, property name prefixed with an underscore).
- Do not output anything outside the explanations and tool calls, and the
initial translation.
````

Finally, the prompt includes a complete example demonstrating the expected transformation from puzzle instance to tool calls.

~~~
The following is a concrete example of how to transform a puzzle into
↪  function
calls. Consider this puzzle:
=== PUZZLE ===
There are 3 houses, numbered 1 to 3 from left to right, as seen from the
front.
Each house contains a different pet.
Each house has a unique attribute for each of the following
↪  characteristics:
- Each pet is a unique type: Cat, Dog, Bird
- They all have a unique breed: Siamese, Labrador Retriever, African Grey
Parrot
- Each pet has a favorite toy: `ball`, `yarn`, `mirror`
- Pets have unique fur colors: `black`, `white`, `spotted`

### Clues:
1. The labrador retriever is in the first house.
2. The bird is not in the third house.
3. The pet whose favorite toy is yarn is in the second house.
4. The siamese is next to the black one.
```


Consider the following output:

```
Harmonized puzzle:

=== PUZZLE ===
There are 3 houses, numbered 1 to 3 from left to right, as seen from the
front.
Each house contains a different pet.
Each house has a unique attribute for each of the following
↪  characteristics:
- Each pet is a unique type: pet_cat, pet_dog, pet_bird
- They all have a unique breed: breed_siamese, breed_labrador-retriever,
breed_african-grey-parrot
- Each pet has a favorite toy: toy_ball, toy_yarn, toy_mirror
- Pets have unique fur-color: fur-color_black, fur-color_white,
↪  fur-color_spotted
### Clues:
1. The pet_labrador-retriever is in the first house.
2. The pet_bird is not in the third house.
3. The pet whose favorite toy is toy_yarn is in the second house.
4. The breed_siamese is next to the fur-color_black one.

Explanation: Initializing 3-house puzzle with 4 property groups and 4
↪  clues.
<tool_call>
{
  "name": "init_zebra",
  "arguments": {
    "number_of_places": 3,
    "number_of_clues": 4,
~~~

```
    "property_groups": {"pet": ["pet_cat", "pet_dog", "pet_bird"],
    ↪  "breed":
["breed_siamese", "breed_labrador-retriever",
↪  "breed_african-grey-parrot"],
"toy": ["toy_ball", "toy_yarn", "toy_mirror"], "fur-color":
↪  ["fur-color_black",
"fur-color_white", "fur-color_spotted"]}
  }
}
</tool_call>

Explanation: The breed_labrador-retriever is in the first house.
<tool_call>
{
  "name": "clue_property_at_house",
  "arguments": {"prop": "breed_labrador-retriever", "house": 1}
}
</tool_call>

Explanation: The pet_bird is not in the third house.
<tool_call>
{
  "name": "clue_not_property_at_house",
  "arguments": {"prop": "pet_bird", "house": 3}
}
</tool_call>

Explanation: The pet whose favorite toy is toy_yarn is in the second
↪  house.
<tool_call>
{
  "name": "clue_property_at_house",
  "arguments": {"prop": "toy_yarn", "house": 2}
}
</tool_call>

Explanation: The breed_siamese is next to the fur-color_black one.
<tool_call>
{
  "name": "clue_property_next_to_property",
  "arguments": {"prop1": "breed_siamese", "prop2": "fur-color_black"}
}
</tool_call>
```

E ANALYSIS OF FAILURE TRACES

We categorize the 33 observed failures of the best-performing model (gpt-oss) on the S, M, and L puzzle instances into two broad classes corresponding to distinct system components: failures of the neural language module and failures of the symbolic reasoning module. The majority of errors (27 cases, or 82%) stem from the neural language module, which converts natural language clues into symbolic tool calls. Within this category, 11 cases (33%) arise from inherently ambiguous clues that would challenge even a human (e.g., ambiguity over whether 'cat' refers to an animal or a pet; see Appx E.1), 7 cases (21%) involve the complete omission of reasoning and tool calls, and 9 cases (27%) involve hallucinations or incorrect execution such as missing, duplicated, or mistranslated tool calls (see Appx E.2).

The remaining 6 failures (18%) arise from the symbolic reasoning stage, specifically from errors in reverting the harmonized MINIEXACT output back into the original attribute- and property-space required for evaluation. In these cases, the executed tool calls yield the correct unique solution, but parsing and de-harmonization introduce missing or duplicate assignments, causing an otherwise semantically correct solution to appear incorrect (see Appx E.3).

E.1 Example Failure due to Ambiguous Clues

This trace illustrates a representative failure due to natural language ambiguity, where the neural language module confuses the semantic categories *animal* and *pet*. Although the natural-language puzzle admits a valid solution, the generated symbolic constraints become inconsistent, and the MiniExact solver therefore returns no prediction.

E.1.1 Puzzle Instance

```
=== PUZZLE ===
There are 2 houses, numbered 1 to 2 from left to right, as seen
from across the street.
Each house is occupied by a different person.
Each house has a unique attribute for each of the following
↪  characteristics:
- Each person has a unique name: `Arnold`, `Eric`
- The people keep unique animals: `cat`, `horse`
- People use unique phone models: `samsung galaxy s21`, `iphone 13`
- Each person has a unique type of pet: `cat`, `dog`
- Everyone has a unique favorite cigar: `pall mall`, `prince`


### Clues:
1. Arnold and the person who uses a Samsung Galaxy S21 are next to each
↪  other.
2. The person partial to Pall Mall and Eric are next to each other.
3. The person who owns a dog is directly left of Eric.
4. The cat lover and the person who uses a Samsung Galaxy S21 are next to
↪  each other.
```

E.1.2 Ground-Truth Solution

The puzzle itself is satisfiable:

House	Person	Animal	Pet	Phone	Cigar
1	Arnold	cat	dog	iphone 13	pall mall
2	Eric	horse	cat	samsung galaxy s21	prince

E.1.3 Symbolic Domain Initialization

After harmonization, the language module initializes the symbolic domain with the following tool call (simplified for readability):

```
init_zebra(
  number_of_places = 2,
  number_of_clues = 4,
  property_groups = {
    person = {person_arnold, person_eric},
    animal = {animal_cat, animal_horse},
    phone  = {phone_samsung-galaxy-s21, phone_iphone-13},
    pet    = {pet_cat, pet_dog},
    cigar  = {cigar_pall-mall, cigar_prince}
  }
)
```

E.1.4 Clues and Generated Tool Calls

The following table shows, how the language module transformed the natural language clues into tool calls.

Natural-language clue	Generated tool call
Arnold and the person who uses a Samsung Galaxy S21 are next to each other.	`clue_property_next_to_property(person_arnold, phone_samsung-galaxy-s21)`
The person partial to Pall Mall and Eric are next to each other.	`clue_property_next_to_property(cigar_pall-mall, person_eric)`
The person who owns a dog is directly left of Eric.	`clue_property_directly_left_of_property(pet_dog, person_eric)`
The cat lover and the person who uses a Samsung Galaxy S21 are next to each other.	**`clue_property_next_to_property(pet_cat, phone_samsung-galaxy-s21)`**

Ambiguity The phrase *"cat lover"* of the 4th clue is semantically ambiguous in this domain: it could refer either to *(i)* the person whose **pet** is a cat (`pet_cat`), or *(ii)* the person who keeps the **animal** cat (`animal_cat`). The model selects `pet_cat`, but the ground-truth interpretation in the puzzle is `animal_cat`.

Under the incorrect interpretation of the tool calls, this implies:

$$\text{Arnold has } pet\_cat \quad \text{and} \quad pet\_dog,$$

which violates the uniqueness constraint for pets per house. Hence the set of constraints become unsatisfiable and MINIEXACT returns no solution.

This example demonstrates how category confusion can propagate into hard symbolic contradictions despite an otherwise valid puzzle.

E.2 EXAMPLE FAILURE DUE TO HALLUCINATION

This trace showcases a failure caused by a simple hallucination in the neural language module. The model generates a symbolic value (`flower_white`) in a tool call that does not exist in the initialized domain, leading to an unsatisfiable set of constraints. Similar errors include duplicating a tool call and omitting another, which also prevents correct symbolic reasoning.

E.2.1 PUZZLE INSTANCE

```
=== PUZZLE ===
There are 6 houses, numbered 1 to 6 from left to right, as seen from
 ↪  across the street.
Each house is occupied by a different person. Each house has a unique
 ↪  attribute for
each of the following characteristics:
 - Each person has a unique name: `Eric`, `Bob`, `Alice`, `Arnold`,
  ↪  `Carol`, `Peter`
 - The people are of nationalities: `swede`, `norwegian`, `german`,
  ↪  `chinese`,
 `dane`, `brit`
 - They all have a unique favorite flower: `daffodils`, `roses`,
  ↪  `carnations`,
 `lilies`, `tulips`, `iris`
 - Each person has a unique favorite drink: `water`, `tea`, `root beer`,
  ↪  `coffee`,
 `boba tea`, `milk`
 - Each person has a favorite color: `green`, `blue`, `purple`, `red`,
  ↪  `yellow`,
 `white`
 - Each person has a unique level of education: `high school`,
  ↪  `associate`, `bachelor`,
 `doctorate`, `trade school`, `master`

### Clues:
1. Arnold is somewhere to the left of the person who loves a carnations
 ↪  arrangement.
```

```
2. The coffee drinker is the person who loves the vase of tulips.
3. The German is directly left of the one who only drinks water.
4. The person whose favorite color is green is Alice.
5. The person who loves white is the coffee drinker.
6. The coffee drinker is the person with a bachelor's degree.
... (remaining clues omitted for brevity)
```

E.2.2 SYMBOLIC DOMAIN INITIALIZATION

```
init_zebra(
  number_of_places = 6,
  number_of_clues = 25,
  property_groups = {
    name        = {name_eric, name_bob, name_alice, name_arnold,
    ↪  name_carol, name_peter},
    nationality = {nationality_swede, nationality_norwegian,
    ↪  nationality_german,
    nationality_chinese, nationality_dane, nationality_brit},
    flower      = {flower_daffodils, flower_roses, flower_carnations,
    ↪  flower_lilies,
    flower_tulips, flower_iris},
    drink       = {drink_water, drink_tea, drink_root-beer, drink_coffee,
    drink_boba-tea, drink_milk},
    color       = {color_green, color_blue, color_purple, color_red,
    ↪  color_yellow,
    color_white},
    edu         = {edu_high-school, edu_associate, edu_bachelor,
    ↪  edu_doctorate,
    edu_trade-school, edu_master}
  }
)
```

E.2.3 CLUES AND GENERATED TOOL CALLS

Natural-language clue	**Generated tool call**
...	...
The person whose favorite color is green is Alice.	`clue_property_at_property(color_green, name_alice)`
The person who loves white is the coffee drinker.	**`clue_property_at_property(flower_white, drink_coffee)`**
The coffee drinker is the person with a bachelor's degree.	`clue_property_at_property(drink_coffee, edu_bachelor)`
...	...

Hallucination The model introduces `flower_white` in a clue even though the valid initialized domain for `flower` only contains: `flower_daffodils, flower_roses, flower_carnations, flower_lilies, flower_tulips, flower_iris`.

This hallucinated symbol cannot be used by the solver, resulting in no prediction. This demonstrates a simple but critical failure mode: the neural language module invents a value, breaking the symbolic reasoning pipeline.

E.3 EXAMPLE FAILURE DUE TO PARSING ERROR

The following trace presents a representative failure due to an error in the post-processing step that converts the symbolic solver output of MINIEXACT into the final structured solution. The symbolic constraints and solver result are correct and match the ground-truth solution, but the parsed solution contains missing and duplicated attribute values.

E.3.1 PUZZLE INSTANCE

```
=== PUZZLE ===
```

```
There are 2 houses, numbered 1 to 2 from left to right, as seen
from across the street. Each house is occupied by a different person.
Each house has a unique attribute for each of the following
↪  characteristics:
 - Each person has a unique name: `Eric`, `Arnold`
 - Each person has a unique level of education: `high school`, `associate`
 - People have unique heights: `short`, `very short`
 - Each person lives in a unique style of house: `victorian`, `colonial`
 - Each person has a favorite color: `yellow`, `red`
 - People have unique favorite sports: `soccer`, `basketball`

### Clues:
1. The person who loves yellow is not in the second house.
2. The person residing in a Victorian house is the person with a high
↪  school diploma.
3. The person with an associate's degree is directly left of the person
↪  with a
high school diploma.
4. The person living in a colonial-style house is the person who loves
↪  soccer.
5. Eric is the person who is short.
6. The person who is very short is the person whose favorite color is
↪  red.
```

E.3.2 GROUND-TRUTH SOLUTION

House	Name	Education	Height	House style	Color	Sport
1	Eric	associate	short	colonial	yellow	soccer
2	Arnold	highschool	veryshort	victorian	red	basketball

E.3.3 HARMONIZED PUZZLE

The natural-language puzzle is first converted into the harmonized representation:

```
=== PUZZLE ===
There are 2 houses, numbered 1 to 2 from left to right, as seen
from across the street. Each house is occupied by a different person.
Each house has a unique attribute for each of the following
↪  characteristics:
 - Each person has a unique name: name_eric, name_arnold
 - Each person has a unique level of education: education_high-school,
 education_associate
 - People have unique heights: height_short, height_very-short
 - Each person lives in a unique style of house: house_style_victorian,
 house_style_colonial
 - Each person has a favorite color: favorite_color_yellow,
  ↪  favorite_color_red
 - People have unique favorite sports: favorite_sport_soccer,
  ↪  favorite_sport_basketball

### Clues:
1. The person who loves favorite_color_yellow is not in the second house.
2. The person residing in a house_style_victorian is the person with a
education_high-school diploma.
3. The person with an education_associate degree is directly left of the
↪  person with a
education_high-school diploma.
4. The person living in a house_style_colonial is the person who loves
favorite_sport_soccer.
5. name_eric is the person who is height_short.
6. The person who is height_very-short is the person whose favorite color
↪  is
favorite_color_red.
```

E.3.4 SYMBOLIC DOMAIN INITIALIZATION

```
init_zebra(
  number_of_places = 2,
  number_of_clues = 6,
  property_groups = {
    name           = {name_eric, name_arnold},
    education      = {education_high-school, education_associate},
    height         = {height_short, height_very-short},
    house_style    = {house_style_victorian, house_style_colonial},
    favorite_color = {favorite_color_yellow, favorite_color_red},
    favorite_sport = {favorite_sport_soccer, favorite_sport_basketball}
  }
)
```

E.3.5 CLUES AND GENERATED TOOL CALLS

Harmonized clue	**Generated tool call**
Harmonized clue 1	`clue_not_property_at_house(favorite_color_yellow, 2)`
Harmonized clue 2	`clue_property_at_property(house_style_victorian, education_high-school)`
Harmonized clue 3	`clue_property_directly_left_of_property(education_associate, education_high-school)`
Harmonized clue 4	`clue_property_at_property(house_style_colonial, favorite_sport_soccer)`
Harmonized clue 5	`clue_property_at_property(name_eric, height_short)`
Harmonized clue 6	`clue_property_at_property(height_very-short, favorite_color_red)`

The generated tool calls are consistent.

E.3.6 MINIEXACT OUTPUT

The symbolic solver returns the following property–house assignments:

```
House 1:
  name           = name_eric
  education      = education_associate
  height         = height_short
  house_style    = house_style_colonial
  color          = favorite_color_yellow
  sport          = favorite_sport_soccer

House 2:
  name           = name_arnold
  education      = education_high-school
  height         = height_very-short
  house_style    = house_style_victorian
  color          = favorite_color_red
  sport          = favorite_sport_basketball
```

This semantically corresponds exactly to the ground-truth solution.

E.3.7 PARSED PREDICTION

Parsing the MINIEXACT output into the final structured representation evaluation results in:

House	Name	Education	Height	House style	Color	Sport
1	Eric	associate	short	colonial	yellow	soccer
2	Arnold	highschool	veryshort	victorian	**/**	**soccer**

Before parsing, the symbolic solution is semantically equivalent to the ground truth. During parsing:

- `soccer` is written to the second house, overwriting `basketball`,
- the color entry for house 2 is omitted.

Thus the corruption is introduced *after* correct symbolic reasoning. Post-processing introduces missing and duplicated attribute assignments, causing an otherwise correct result to be evaluated as incorrect.