

# Streamliners for Answer Set Programming

Florentina Voboril

TU Wien  
Vienna, Austria

`florentina.voboril@tuwien.ac.at`

Martin Gebser

University of Klagenfurt  
Klagenfurt, Austria

`Martin.Gebser@aau.at`

Stefan Szeider

TU Wien  
Vienna, Austria

`sz@ac.tuwien.ac.at`

Alice Tarzariol

University of Klagenfurt  
Klagenfurt, Austria

`Alice.Tarzariol@aau.at`

Streamliner constraints reduce the search space of combinatorial problems by ruling out portions of the solution space. We adapt the StreamLLM approach, which uses Large Language Models (LLMs) to generate streamliners for Constraint Programming, to Answer Set Programming (ASP). Given an ASP encoding and a few small training instances, we prompt multiple LLMs to propose candidate constraints. Candidates that cause syntax errors, render satisfiable instances unsatisfiable, or degrade performance on all training instances are discarded. The surviving streamliners are evaluated together with the original encoding, and we report results for a virtual best encoding (VBE) that, for each instance, selects the fastest among the original encoding and its streamlined variants. On three ASP competition benchmarks (Partner Units Problem, Sokoban, Towers of Hanoi), the VBE achieves speedups of up to  $4\text{--}5\times$  over the original encoding.

## 1 Introduction

Answer Set Programming (ASP) has emerged as a powerful paradigm for declarative problem solving, with successful applications in planning, scheduling, configuration, and knowledge representation [3]. Modern ASP solvers like CLINGO [13] combine grounding with conflict-driven clause learning, enabling the solution of hard combinatorial problems. However, solver performance remains highly sensitive to problem encodings: two logically equivalent formulations can lead to vastly different grounding sizes and solving times.

A well-known technique for improving solver performance is the addition of *streamliner constraints* [15]. These are auxiliary constraints intended to reduce the search space by ruling out parts of the solution space. They need not be logically implied by the original encoding and may remove some solutions, but should preserve satisfiability for the instances of interest. Streamliners include *symmetry-breaking constraints* [35], which remove equivalent solutions arising from problem symmetries, and *implied constraints*, which make implicit problem structure explicit to the solver. In the Constraint Programming (CP) community, Spracklen et al. [24] recently demonstrated that portfolios of automatically generated streamliners can match or exceed hand-crafted constraints. In ASP, systems like SBASS [9] and BREAKID [7] automatically generate propositional symmetry-breaking constraints, while inductive logic programming methods such as conflict-driven constraint learning [18] and core-guided reformulation [19] derive first-order constraints from small training instances [27]. These methods either operate at the propositional level (often causing grounding bottlenecks on larger instances) or require carefully designed language biases (limiting generality).

Voboril et al. [34] recently demonstrated that Large Language Models (LLMs) can generate streamliner constraints for CP models. Their *StreamLLM* approach prompts an LLM with a problem encoding

and asks it to propose constraints that reduce the search space. Candidate constraints are validated on small training instances and the best-performing ones are retained. This approach requires no hand-crafted language bias or domain templates, but only uses the encoding plus a few small training instances.

The combination of LLMs and ASP has been explored along two directions. Neural-symbolic hybrids like NeurASP [38] and differentiable ASP solvers [23] integrate neural components with ASP reasoning, but assume that the logic program is manually written. For automated program generation, Ishay et al. [17] prompt LLMs to produce complete ASP programs from natural language (NL), and Coppolillo et al. [6] fine-tune LLMs for ASP syntax. However, one-shot generation often yields flawed programs [22]. Agentic approaches address this through iterative refinement with solver feedback: the MCP-Solver [26] exposes ASP capabilities to LLMs via the Model Context Protocol, and evaluation on ASP-Bench [25] shows that feedback-driven agents succeed on benchmark problems. Our work differs from NL-to-ASP translation as we do not generate entire programs, but propose auxiliary constraints that augment existing encodings.

In this paper, we investigate whether LLM-based streamliner generation can be adapted to ASP, which presents distinct challenges than CP: the non-monotonic semantics of stable models means that adding constraints or auxiliary definitions can have subtle effects on solution sets. LLMs may generate not only constraints but also helper rules that introduce new atoms, which can interact non-trivially with the minimality condition of stable models. The rich syntax of ASP (aggregates, choice rules, conditional literals, optimization statements) also requires LLMs to generate syntactically valid and semantically meaningful code. We adapt the StreamLLM pipeline to ASP by (1) prompting multiple frontier LLMs to generate candidate streamliners for a given ASP encoding, (2) filtering candidates that cause syntax errors, make satisfiable training instances unsatisfiable, or degrade runtime performance on all training instances, (3) selecting the best-performing streamliners based on runtime improvements, and (4) evaluating the original encoding together with the streamlined variants as well as the virtual best encoding (VBE) for each instance, which serves as an oracle-style analysis of complementarity.

We evaluate our approach on three benchmark problems from the ASP competition 2011 [4] - Partner Units Problem, Sokoban, and Towers of Hanoi - observing speedups of up to 4–5 $\times$  over the original encoding. The observation that different LLMs and different runs produce semantically diverse constraints, not simple syntactic variations, suggests that our approach captures genuine problem structure rather than exploiting superficial patterns.

The rest of this paper is structured as follows. Section 2 provides background on ASP and related work. Section 3 describes our adaptation of StreamLLM to ASP. Section 4 presents the experiments and analyzes the generated streamliners. Section 5 concludes by discussing limitations and future directions.

## 2 Preliminaries

This section provides brief overviews of ASP and previous works related to streamliner constraints.

### 2.1 Answer Set Programming

We consider first-order ASP programs in the modeling language of CLINGO [3, 12]. That is, *terms* include (i) *constants*, which can be tokens starting with a lowercase letter, integers, strings enclosed in double quotes, the least term  $\#_{\text{inf}}$  or the greatest term  $\#_{\text{sup}}$ , (ii) *variables*, i.e., tokens starting with an uppercase letter or the anonymous variable “\_”, and (iii) *functions* of the form  $f(t_1, \dots, t_n)$ , for a function symbol  $f$  (a token starting with a lowercase letter) and terms  $t_1, \dots, t_n$ , tuples  $(t_1, \dots, t_n)$ , or arithmetic

expressions composed of integers, variables, operators “+”, “-”, “\*”, “/”, etc. and parentheses.

*Atoms* can be of three forms: (i) *symbolic atoms*  $p(t_1, \dots, t_n)$ , for a predicate symbol  $p$  (a token starting with a lowercase letter) and terms  $t_1, \dots, t_n$ , which are just written  $p$  in case  $n = 0$ , (ii) *built-in atoms*  $t_1 \circ t_2$  composed of terms  $t_1, t_2$  and a comparison operator  $\circ \in \{<, <=, >=, >, =, !=\}$ , and (iii) *aggregate atoms*  $t_1 \circ_1 \#aggr\{e_1; \dots; e_k\} \circ_2 t_2$ , where  $t_1, t_2$  are terms,  $\circ_1, \circ_2$  are comparison operators,  $\#aggr \in \{\#count, \#sum, \#min, \#max\}$  is an *aggregate function*, and each  $e_i$ , for  $1 \leq i \leq k$ , is an *aggregate element*  $t_{1_i}, \dots, t_{n_i} : \ell_{1_i}, \dots, \ell_{m_i}$  over terms  $t_{1_i}, \dots, t_{n_i}$  and *basic literals*  $\ell_{1_i}, \dots, \ell_{m_i}$  such that each  $\ell_{j_i}$ , for  $1_i \leq j_i \leq m_i$ , is  $a$ ,  $\text{not } a$  or  $\text{not not } a$  for some symbolic or built-in atom  $a$ . Basic literals as well as  $a$ ,  $\text{not } a$  and  $\text{not not } a$  over an aggregate atom  $a$  are *literals*. A *choice element*  $c_i$  is an aggregate element  $t_{1_i} : \ell_{1_i}, \dots, \ell_{m_i}$  such that the (constant or function) term  $t_{1_i}$  has the form of a symbolic atom, and  $t_1 \circ_1 \{c_1; \dots; c_k\} \circ_2 t_2$  with choice elements  $c_1, \dots, c_k$  is a *choice*. Either or both of the comparison operators  $\circ_1, \circ_2$  can be omitted for a choice or an aggregate atom, in which case they default to “<=”, and if  $t_1$  or  $t_2$  is skipped in addition,  $\#inf$  resp.  $\#sup$  is taken as the corresponding default value.

An ASP program is a set of *rules* of the form  $h :- b_1, \dots, b_m$ , where  $h$  is a symbolic atom or a choice, and  $b_1, \dots, b_m$  are literals. The rule head  $h$  may be omitted, in which case we call the rule a *constraint*. In order to be processed by CLINGO, an ASP program has to be *safe* [3], roughly meaning that all variables must have positive occurrences (outside of arithmetic expressions) in symbolic atoms belonging to the rule body  $b_1, \dots, b_m$ . The semantics of an ASP program is given by *stable models* [10], which are sets of (true) ground atoms that satisfy and are derivable from the instances of the program’s (first-order) rules. We say that an ASP program is *satisfiable* if it has some stable model, and *unsatisfiable* otherwise. Proper first-order rules are called an ASP *encoding*, and the rules without variables a problem *instance*.

## 2.2 Related Works

Although the term streamliner is not widely used in the ASP literature, many works have investigated the identification of constraints that prune the search space to improve solving performance. Regarding symmetry breaking, research in ASP has focused on automatically producing *symmetry-breaking constraints* (SBCs), inspired by techniques from the Boolean satisfiability (SAT) community that generate propositional SBCs based on properties of permutation groups [21]. The systems SBASS [9] and BREAKID [7] analyse a ground ASP program to identify symmetries and add propositional SBCs, which can speed up solving on small and highly combinatorial instances, but often does not scale to large ones. The work of Devriendt et al. [8] analyses predicates for interchangeable argument values and adds propositional SBCs imposing lexicographical orderings; however, this approach handles only local domain symmetries. Moreover, propositional SBCs can negatively affect solving performance through redundant constraints, and users gain no insight into the symmetry structure.

These shortcomings are addressed by Tarzariol et al. [27], who exploit symmetries identified by SBASS to define an inductive logic programming task that learns first-order constraints for both decision problems [29] and optimization problems [28]. The constraints are obtained by lifting propositional symmetries from small representative instances and are validated on a test set, but without formally proving them; thus, this approach generates streamliners that are not necessarily SBCs. The use of inductive logic programming requires providing a language bias specifying building blocks for the constraints, which makes learning efficient constraints almost as challenging as optimizing an encoding by hand.

Domain-specific streamliners have been developed for particular applications: Yli-Jyrä et al. [39] incorporate pruning constraints into preventive maintenance scheduling, while Husár et al. [16] design pruning strategies for multi-agent pathfinding on large instances. Cappanera et al. [5] adopt logic-based Benders decomposition for healthcare scheduling. These approaches rely on domain expertise and do

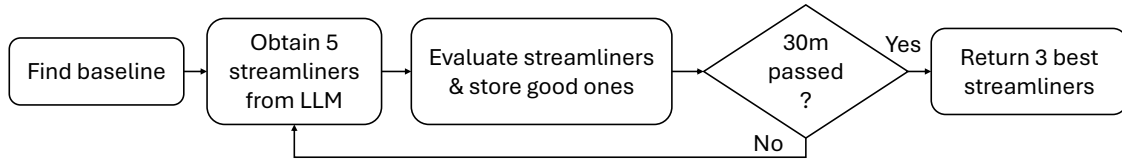


Figure 1: Fully automated pipeline of the approach

not transfer to other problems.

Studies on generalising conflict clauses [14] have been conducted for ASP encodings [37, 11, 30] and temporal domains [20]. Our approach differs in not being limited to streamliners derived from a particular constraint type (e.g., conflict clauses), but rather generating constraints from diverse sources.

### 3 Streamlining ASP

The pipeline in Figure 1 illustrates our approach, which is based on StreamLLM [34]. It runs fully automated without human interaction. As input, it requires an ASP encoding together with a few satisfiable training instances that can be solved within a few seconds. First, we run the original encoding on the training instances to obtain baseline running times. Then, we provide the LLM with the encoding and the prompt in Figure 2.

The prompt is divided into three components: first, it states the objective, namely, to generate five additional CLINGO constraints that improve solving efficiency by eliminating symmetries or redundant solutions. Second, it provides a step-by-step description of how the LLM should analyze the encoding and derive suitable constraints. This procedural guidance follows the principles of the Chain-of-Thought technique [36]. Finally, the prompt specifies compliance rules that require syntactic correctness of the generated CLINGO code and encourage diversity and creativity in the proposed constraints. This prompt is similar to the one used for the original StreamLLM approach. The main differences are the following:

- Our approach asks for CLINGO constraints, while the original approach uses MiniZinc.
- We only ask for well-studied streamliners in ASP, namely symmetry-breaking and implied constraints. In contrast, the original StreamLLM approach is also interested in streamliners that might make some instances unsatisfiable.
- The prompt for the original StreamLLM approach tells the LLM how to deal with feedback on previously provided constraints. As the experiments by Voboril et al. [34] showed that this adaptive variant is not working significantly better than a variant without feedback, we omit this part.

Typically, the LLM returns single constraints. However, it is also possible that it returns multiple constraints or new rules. In our procedure, we treat them the same as a single constraint.

The obtained streamliners are individually added to the original encoding and tested on the training instances independently. A streamliner that leads to syntax errors, unsatisfiability, or has a longer running time than the original encoding on all training instances is discarded. The remaining streamliners that improve the running time on at least one training instance are stored. We repeat this procedure for 30 minutes. After that, the pipeline stops and returns the three streamliners that form the best combination under a VBE criterion on the training instances. Specifically, for each training instance, we consider the

**Objective**

Analyze the given ASP problem in Clingo and suggest five additional symmetry-breaking or implied constraints to enhance the problem-solving process.

**Steps**

1. **Analyze Content:** Read the provided ASP encoding. Understand the problem being addressed, including its facts, predicates, rules and constraints.
2. **Generate additional constraints:** Based on your analysis, create five unique constraints. These should eliminate equivalent solutions in a search space to reduce redundancy and improve solver efficiency.
3. **Always return your constraints as a JSON object,** adhering to the structure: {"constraint\_1": "<Clingo constraint>", ..., "constraint\_5": "<Clingo constraint>"}. Your final output should exclusively be the JSON object containing the five constraints. **DO NOT** include any comments.

**Compliance Rules**

1. **Code Quality:** All Clingo code provided must be syntactically correct and functional.
2. **Creativity:** You're encouraged to be innovative in proposing constraints, keeping in mind their purpose: to narrow down the search space efficiently by eliminating equivalent solutions.

Figure 2: Used prompt

minimum running time across the original encoding and the selected streamliners, and the chosen triple minimizes the sum of these per-instance minimum running times.

The selected streamliners can then be applied to larger instances. For our experiments, we individually run the three streamlined encodings as well as the original encoding with all three streamliners, and we report the VBE, i.e., the fastest running time achieved by the original encoding or any of the streamlined variants on each instance. This allows us to analyze the extent to which different streamliners provide complementary performance improvements.

## 4 Experiments

The experiments were conducted on compute nodes with 2.40GHz, 10-core 2×Intel Xeon E5-2640 v4 processors. We used CLINGO 5.8.0 as the ASP solver, which was invoked via its Python API with Python 3.11.5. In every iteration, we randomly decided for one of the following LLMs: Claude 4.5 Sonnet (anthropic/claude-sonnet-4.5), GPT-5 Mini (openai/gpt-5-mini), Gemini 3 Pro (google/gemini-3-pro-

```

% Constraint P1
:- comUnit(U), #count{Z: unit2zone(U,Z)} = 0, #count{S: unit2sensor(U,S)} > 0.

% Constraint P2
:- unit2sensor(U1,S), unit2sensor(U2,S), U1 > U2.

% Constraint P3
:- unit2zone(U+1, _), not unit2zone(U, _), comUnit(U), comUnit(U+1).

% Constraint P4
:- comUnit(U), unit2zone(U,Z1), unit2zone(U,Z2), Z1 < Z2,
   #count{S: zone2sensor(Z1,S), zone2sensor(Z2,S)} = 0.

% Constraint P5
:- unit2sensor(U1,S), unit2sensor(U2,S), U1 > U2.

% Constraint P6
:- comUnit(U1), comUnit(U2), U1 < U2, sensor(S),
   S = #min{S': sensor(S')}, unit2sensor(U2,S), #count{Z: unit2zone(U1,Z)} = 0.

```

Listing 1: PUP Streamliners

preview), Mistral Large (mistralai/mistral-large-2512), and Deepseek V3.2 (deepseek/deepseek-v3.2). All models were accessed via the OpenRouter API using identical prompts and generation parameters to ensure comparability across runs. Our repository [32] provides the original and streamlined encodings, instances, and a table of running times for each problem, together with the code used for our approach.

For testing our method, we investigate its application on three combinatorial problems derived from the 2011 ASP competition [4], namely, *Partner Units Problem* (PUP), *Sokoban* and *Towers of Hanoi*, focusing on the decision version of these problems. Our results compare three individually streamlined encodings, their combined version, the original encoding, and a VBE that picks the fastest encoding per instance. For each problem, we did two runs to examine how much the streamliners generated by our approach vary. To account for unsolved instances, we aggregate running times using penalized average running times. Specifically, timeouts are penalized by a factor of 2 (PAR2), where a timeout corresponds to 20 minutes. Instances on which all encodings led to timeouts are excluded from the aggregated running times to avoid large offsets due to their PAR2 times. In the following subsections, we present and discuss the results for each problem and overall.

#### 4.1 Partner Units Problem

The Partner Units Problem (PUP) is an abstract representation of configuration problems occurring in railway safety or building security systems [31]. Its input consists of a set  $U$  of units and a bipartite graph  $G = (S, Z, E)$ , where  $S$  is a set of sensors,  $Z$  is a set of security/safety zones, and  $E$  is a relation between  $S$  and  $Z$ . The task is to find an assignment of vertices  $v \in S \cup Z$  to units  $U$  such that the following conditions hold for each unit and integers  $UCAP$  and  $IUCAP$ : (i) each unit contains at most  $UCAP$  many sensors and  $UCAP$  many zones; and (ii) each unit has at most  $IUCAP$  adjacent units, where the units  $u_1 \neq u_2$  are adjacent whenever  $v_i \in u_1$  and  $v_j \in u_2$  for some  $(v_i, v_j) \in E$ . For this problem, we used the benchmark suite by Aschinger et al. [1], focusing on the *double* instance collections with  $UCAP = IUCAP = 2$ , of which we extracted 7 instances for training and 26 instances for testing.

Listing 1 shows the streamliners that we obtained with our approach in two runs, the former pro-

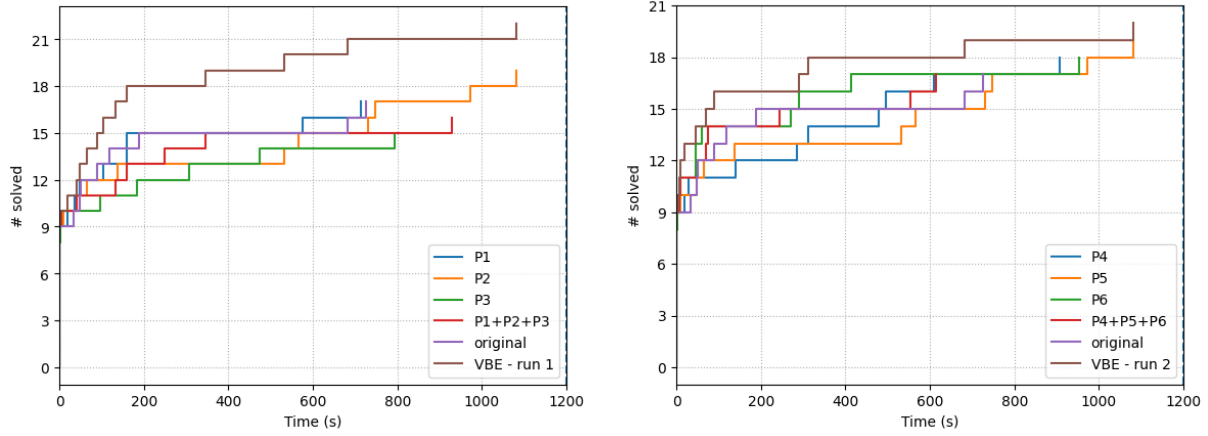


Figure 3: Results for PUP for run 1 (left) and run 2 (right)

Table 1: Sum of aggregated running times in seconds for all PUP instances

|              | <b>P1</b> | <b>P2</b> | <b>P3</b> | <b>P1+P2+P3</b> | <b>original</b> | <b>VBE</b> |
|--------------|-----------|-----------|-----------|-----------------|-----------------|------------|
| <i>run 1</i> | 13,813    | 12,091    | 18,660    | 16,265          | 13,938          | 3,302      |
|              | <b>P4</b> | <b>P5</b> | <b>P6</b> | <b>P4+P5+P6</b> | <b>original</b> | <b>VBE</b> |
| <i>run 2</i> | 12,880    | 12,091    | 11,696    | 13,627          | 13,938          | 7,412      |

ducing constraints P1, P2, and P3 and the latter producing constraints P4, P5, and P6. Note that the constraints P2 and P5, obtained in separate runs, coincide and are redundant in the sense that they reassert a condition already implied by the given encoding: no sensor can be assigned to multiple units. The constraints P3 and P6 break symmetries on the assignment of sensors and zones to units, without impairing satisfiability because the identifiers/integers for the units in  $U$  are interchangeable. Unlike that, the constraints P1 and P4 impose the non-trivial conditions that each unit must contain some zone or that two zones belonging to the same unit must have some sensor in common, respectively. These two streamliners preserve the satisfiability of our training instances (and the test instances as well), while their general validity is not obvious and would necessitate a formal proof.

Figure 3 displays two cactus plots showing the performance obtained on the test set by the constraints resulting from two runs of our approach. Each plot shows the number of instances solved by (i) the baseline encoding (original), (ii) the baseline encoding combined with each single streamliner constraint, (iii) the baseline encoding combined with all three constraints produced in the same run, and (iv) the VBE, i.e., the best-performing result per instance across all considered encodings. While the VBE could solve 22 resp. 20 instances, the original encoding could only solve 17 instances within the time limit. This indicates that different streamlined encodings contribute complementary strengths, as no single encoding strictly dominates across all instances. Table 1 shows the accumulated running times for the two runs, using PAR2 for timeouts. In particular, we observe that most constraints improve the accumulated running time on the test instances. Only P3 performed worse overall than the original encoding, since it is effective on a few test instances only. The VBE indicates a potential running time reduction by 76% (resp. 47% in run 2). Out of the 26 test instances, four instances could not be solved with any encoding and we exclude their PAR2 times in Table 1.

```

% Constraint S1
:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T), X1 < X3.

% Constraint S2
:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T), X1 = X3, Y1 < Y3.

% Constraint S3
:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T+1), X1+Y1 > X3+Y3,
   not box(X2,Y2,T), X2 != X3, Y2 != Y3.

% Constraint S4
:- push(X1,Y1,X2,Y2,DX,DY,T), push(X2,Y2,X1,Y1,ODX,ODY,T+1).

% Constraint S5
:- push(X1,Y1,X2,Y2,DX,DY,T), push(X1,Y1,X3,Y3,DX,DY,T), X2 = X3, Y2 < Y3.

% Constraint S6
:- push(X1,Y1,X2,Y2,1,0,T), push(X3,Y3,X4,Y4,-1,0,T), X2 = X4, Y2 < Y4.

```

Listing 2: Sokoban Streamliners

**Lifting Symmetry Breaking with Inductive Logic Programming.** The work of Tarzariol et al. [29] has a similar target to our approach, namely, to learn first-order constraints preserving the satisfiability of instances. In particular, one of the problems tackled is PUP, where the instances of the family *double* showed more difficulty, specifically on satisfiable instances. The constraints obtained in the work of Tarzariol et al. [29] outperformed our VBE for the first 15 test instances, yielding an improvement of 87% over the original encoding, whereas the VBE achieved improvements of 64% (resp. 62% for the second run). However, they did not manage to return any solution within the considered timeout for the remaining 7 instances. When aggregating running times over all instances using PAR2, the approach performs 22% worse than the original encoding. It is worth noting that our constraints were generated by LLMs without specifying any restrictions on their structure, whereas for the work of Tarzariol et al. [29], a language bias must be provided. As a result, the constraints we learned have a richer structure, e.g., aggregates, arithmetic expressions such as the increment of numerical variables, built-in comparison operators like “<”, etc.

## 4.2 Sokoban

Sokoban is a planning problem, where each instance defines the (grid) layout of a warehouse, consisting of walls and locations (possibly designed for storage) that can hold at most one box each, and a starting situation describing the position of the agent, called sokoban, as well as boxes in the warehouse. The sokoban can walk on locations (unless occupied by some box) and push single boxes onto unoccupied locations. A solution of the problem is a sequence of actions to move all boxes onto storage locations. For this problem, we split the benchmark suite by Bomanson et al. [2] into 10 training instances and 18 test instances.

The streamliners obtained in two separate runs, denoted by S1, S2, and S3 or S4, S5, and S6, respectively, are shown in Listing 2. The majority of them, namely, S1, S2, S5, and S6, redundantly assert that different kinds of push actions cannot be performed simultaneously. Similarly, the condition expressed by S3, i.e., the target of a push action must contain a box in the next state, is redundant, where also taking

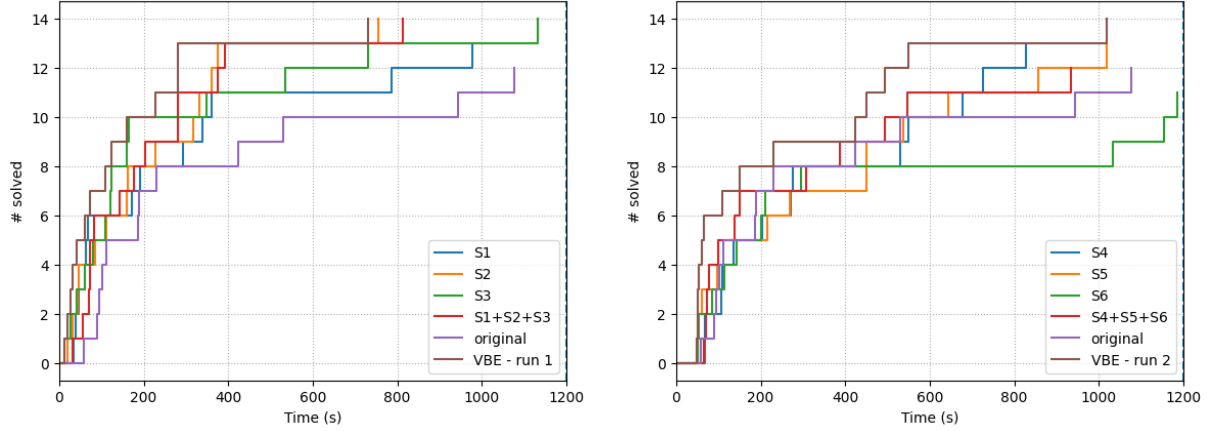


Figure 4: Results for Sokoban for run 1 (left) and run 2 (right)

Table 2: Sum of running times in seconds for all Sokoban instances

|              | <b>S1</b> | <b>S2</b> | <b>S3</b> | <b>S1+S2+S3</b> | <b>original</b> | <b>VBE</b> |
|--------------|-----------|-----------|-----------|-----------------|-----------------|------------|
| <i>run 1</i> | 5,831     | 3,026     | 3,648     | 3,051           | 8,837           | 2,178      |
|              | <b>S4</b> | <b>S5</b> | <b>S6</b> | <b>S4+S5+S6</b> | <b>original</b> | <b>VBE</b> |
| <i>run 2</i> | 6,948     | 7,197     | 11,727    | 8,144           | 8,837           | 3,758      |

the next push action (at time  $T+1$ ) into account is actually unnecessary. While the constraint S4, which states that a push action should not be immediately reversed, may seem straightforward too, it amounts to a satisfiability-preserving policy rather than a strictly necessary condition.

Similar to PUP in the previous subsection, Figure 4 provides two cactus plots showing the performance obtained on the test set by the constraints resulting from two runs of our approach. The original encoding could solve 12 instances within the timeout, while we managed to solve 2 more instances when considering S2, S3, or the combination of all three constraints from the first run. Regarding the second run, only the VBE solved 14 instances. Table 2 shows the accumulated running times in seconds for the instances solved with at least one encoding, thus leaving out four of the 18 test instances. In particular, the table shows that the VBE in the first run achieves a 75% reduction in running time compared to the original encoding, while it yields a 57% reduction in the second run.

### 4.3 Towers of Hanoi

Towers of Hanoi is a puzzle where the input is the initial configuration of  $n$  disks of different sizes on 3 pegs. Usually, in the initial configuration, the disks are all placed on the first peg. The goal is to find a sequence of movements of the disks such that all disks are moved from the first peg to the third one. Only the topmost disk of a peg can be moved at a time, and a disk cannot be moved to a peg already containing some disk of smaller size. We synthetically generated 11 trivial instances for training, while 46 test instances were taken from the work of Gebser et al. [13].

Listing 3 presents the streamliners T1, T2, and T3, obtained in the first run, and the streamliners T4, T5, and T6 from the second run. Notably, T3 is not an actual constraint but merely a comment (thus violating the instructions of Step 3 in Figure 2). Adding such a comment does not affect the

```

% Constraint T1
:- moved(D1,T-1), moved(D2,T), D2 <= D1, disk(D1+1).

% Constraint T2
stable(1,P,T) :- on(1,P,T), goal_on(1,P), time(T).
stable(D,P,T) :- on(D,P,T), goal_on(D,P), stable(D-1,P,T), disk(D), disk(D-1),
                  time(T).
:- move(D,P,T), stable(D,P_prev,T-1), on(D,P_prev,T-1).

% Constraint T3: a comment

% Constraint T4
:- disk(D), move(D,P1,T1), move(D,P2,T2), move(D,P1,T3), T1 < T2, T2 < T3,
   T3 = T2 + 1, P1 = P2.

% Constraint T5
:- move(D1,P1,T), move(D2,P2,T+1), D1 < D2, on(D1,Px,T-1), on(D2,Py,T),
   Px = Py, P1 != Py, P2 = Px.

% Constraint T6
:- time(T), T > 2, move(D,P1,T-1), move(D,P2,T-2), P1 = P2, move(D,P1,T).

```

Listing 3: Towers of Hanoi Streamliners

solving performance. However, due to minor measurement noise in the running times of the training instances, selecting T1 and T2 (along with a comment T3) was apparently better for the performance than selecting a third proper constraint. While the constraints T4, T5, and T6 seem complex, all three reject redundant null moves placing a disk back on its current peg, which is already prohibited by the encoding because the current peg is not a valid target for moving a disk. The constraint T1 is also redundant yet less straightforward by asserting the implied condition that a disk cannot be moved when the two other pegs are already occupied by smaller disks.<sup>1</sup> Interestingly, T2 constitutes a subprogram defining the auxiliary predicate `stable` to identify disks that have reached their goal configuration. Along with a constraint rejecting further moves of such disks, the streamliner T2 imposes the policy to preserve already established parts of the goal configuration.<sup>2</sup>

The cactus plot in Figure 5 and the accumulated running times in Table 3 clearly show that the original encoding performed worse than all streamlined encodings and the VBE in particular. While the original encoding only solved 31 instances within the timeout, the VBE could solve all 46 test instances in both runs. Further, it could reduce the running time by 79% resp. 71% in the second run.

#### 4.4 Analysis of Streamliners

Within the 30 minutes allocated to training, our pipeline (cf. Figure 1) performed between 10 and 29 LLM calls. The amount of LLM calls in each run primarily depends on the number and size of the training instances. Comparing the generated streamliners, 52% resulted in performance improvements on the training instances. In contrast, 19% produced syntax errors, 9% led to unsatisfiability on all training instances, and a further 8% rendered some (but not all) training instances unsatisfiable. The

<sup>1</sup>If  $D1 = D2$ , T1 asserts the policy that a disk should not be moved and immediately placed back on its previous peg.

<sup>2</sup>Similar ideas have been incorporated, e.g., in Blocksworld planning encodings [12], but we are unaware of any Towers of Hanoi encoding in the ASP literature from which the LLM could have directly retrieved T2.

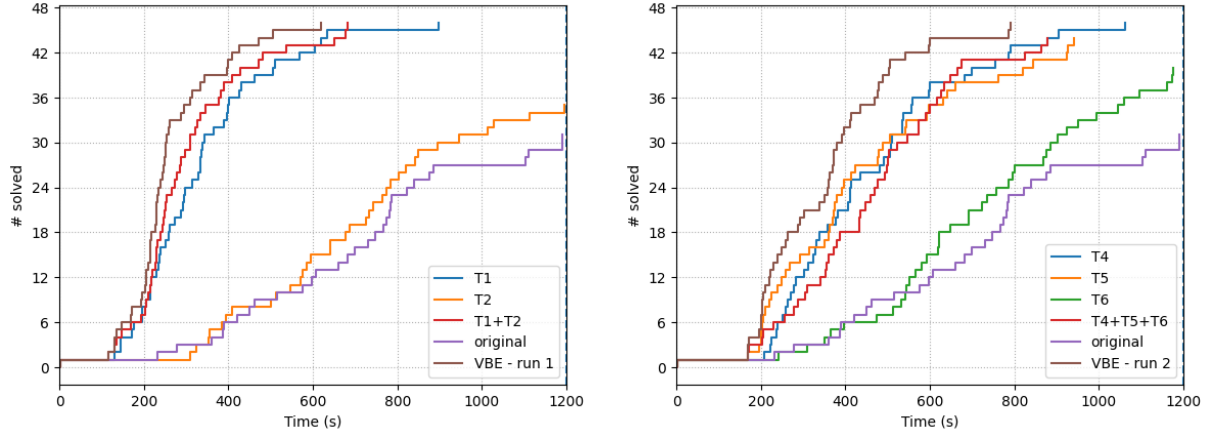


Figure 5: Results for Towers of Hanoi for run 1 (left) and run 2 (right)

Table 3: Sum of running times in seconds for all Towers of Hanoi instances

|              | <b>T1</b> | <b>T2</b> | <b>T3</b> | <b>T1+T2</b>    | <b>original</b> | <b>VBE</b> |
|--------------|-----------|-----------|-----------|-----------------|-----------------|------------|
| <i>run 1</i> | 15,108    | 49,494    | -         | 13,702          | 56,710          | 11,833     |
|              | <b>T4</b> | <b>T5</b> | <b>T6</b> | <b>T4+T5+T6</b> | <b>original</b> | <b>VBE</b> |
| <i>run 2</i> | 20,853    | 23,832    | 42,569    | 24,755          | 56,710          | 16,053     |

remaining 12% neither caused unsatisfiability nor improved upon the baseline performance.

While the selected streamliners empirically improve the performance of CLINGO, at least on the training instances, we find that they often have quadratic grounding size or can be strengthened by dropping unnecessary side conditions. The PUP constraint P2 is an example of quadratic grounding size, as it is formulated over pairs  $u_1 > u_2$  of units. Its manual reformulation, denoted by P2r, achieves linear grounding size by using a `#count` aggregate to express the same condition as P2. Side conditions are, e.g., dropped from the Sokoban constraint S3, where omitting the reference to a second push action yields the stronger, manually revised constraint S3r. Our manually revised versions of generated streamliners, provided whenever the grounding size or side conditions can be reduced, are detailed in our repository [32].

Table 4 contrasts the sum of running times in seconds obtained with the original and our manually revised constraints. Although the revised constraints should have generally better scaling behavior, we observe a deterioration of CLINGO’s running times in almost all cases. This comes unexpectedly and shows that the StreamLLM pipeline effectively selects streamliners that empirically improve the solver performance on the training and then also the test instances. On the one hand, `#count` aggregates in our grounding-oriented revisions can be more demanding to propagate than (a quadratic number of) “flat” constraints. On the other hand, adding redundant constraints that are also implied by a problem encoding may negatively affect the solver’s internal heuristics, so that strengthening them does not necessarily provide better performance. From the negative results for our manually revised constraints, we conclude that the empirical approach of the StreamLLM pipeline manages to tune the solver performance better than general scaling considerations.

Table 4: Comparison of original and revised constraints for PUP, Sokoban, and Towers of Hanoi

|                | <b>P1</b> | <b>P1r</b> | <b>P2</b> | <b>P2r</b> | <b>P6</b> | <b>P6r</b> |           |            |           |            |
|----------------|-----------|------------|-----------|------------|-----------|------------|-----------|------------|-----------|------------|
| <i>PUP</i>     | 13,813    | 13,810     | 12,091    | 13,862     | 11,696    | 12,077     |           |            |           |            |
|                | <b>S1</b> | <b>S1r</b> | <b>S2</b> | <b>S2r</b> | <b>S3</b> | <b>S3r</b> | <b>S5</b> | <b>S5r</b> | <b>S6</b> | <b>S6r</b> |
| <i>Sokoban</i> | 5,831     | 13,578     | 3,026     | 10,101     | 3,648     | 9,010      | 7,197     | 9,532      | 11,727    | 10,422     |
|                | <b>T1</b> | <b>T1r</b> | <b>T2</b> | <b>T2r</b> | <b>T4</b> | <b>T4r</b> | <b>T5</b> | <b>T5r</b> | <b>T6</b> | <b>T6r</b> |
| <i>Hanoi</i>   | 15,108    | 34,236     | 49,494    | 56,211     | 20,853    | 52,795     | 23,832    | 57,188     | 42,569    | 52,795     |

## 5 Conclusion

This paper presented the first approach of LLM-based streamliner generation for ASP. Overall, the results demonstrate that our approach is a viable and powerful tool for improving ASP solving performance. We showed that LLMs can propose syntactically valid and semantically meaningful constraints that effectively prune the search space. By filtering candidate streamliners and selecting the best-performing ones, we obtain complementary constraints. Evaluating the VBE yields runtime reductions of 47%-79%, with most cases exceeding 70%, indicating substantial potential speedups when selecting the best encoding per instance. Moreover, in line with earlier findings by Voboril et al. [34], our results suggest that only a small number of training instances is sufficient to guide effective streamliner generation, which is particularly encouraging for ASP where large curated training sets are costly. Several directions for future work remain. First, our approach should be evaluated on a broader range of ASP benchmarks, including larger instance families and additional domains, to better understand its generality and limits. Second, the quality of generated streamliners may be improved through model adaptation techniques such as fine-tuning or knowledge distillation, as well as more advanced prompt-optimization methods (e.g., by prompt learning or a combination with symbolic methods). Finally, extending the method to optimization problems, in line with recent work by Voboril et al. [33], is a natural and important next step for ASP as well.

## 6 Acknowledgment

This research was funded by the Austrian Science Fund (FWF) projects 10.55776/PAT1599524, 10.55776/COE12 and 10.55776/P36420.

## References

- [1] Markus Aschinger, Conrad Drescher, Gerhard Friedrich, Georg Gottlob, Peter Jeavons, Anna Ryabokon & Evgenij Thorstensen (2011): *Optimization Methods for the Partner Units Problem*. In: *Proceedings of the 8th International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, CPAIOR 2011, Lecture Notes in Computer Science 6697*, Springer, pp. 4–19, doi:10.1007/978-3-642-21311-3\_4.
- [2] Jori Bomanson, Martin Gebser, Tomi Janhunen, Benjamin Kaufmann & Torsten Schaub (2016): *Answer Set Programming Modulo Acyclicity*. *Fundamenta Informaticae* 147(1), pp. 63–91, doi:10.3233/FI-2016-1398.
- [3] Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Marco Maratea, Francesco Ricca & Torsten Schaub (2020): *ASP-*

- Core-2 Input Language Format*. *Theory and Practice of Logic Programming* 20(2), pp. 294–309, doi:10.1017/S1471068419000450.
- [4] Francesco Calimeri, Giovambattista Ianni & Francesco Ricca (2014): *The Third Open Answer Set Programming Competition*. *Theory and Practice of Logic Programming* 14(1), pp. 117–135, doi:10.1017/S1471068412000105.
  - [5] Paola Cappanera, Marco Gavanelli, Maddalena Nonato & Marco Roma (2023): *Logic-Based Benders Decomposition in Answer Set Programming for Chronic Outpatients Scheduling*. *Theory and Practice of Logic Programming* 23(4), pp. 848–864, doi:10.1017/s147106842300025x.
  - [6] Erica Coppolillo, Francesco Calimeri, Giuseppe Manco, Simona Perri & Francesco Ricca (2024): *LLASP: Fine-Tuning Large Language Models for Answer Set Programming*. In: *Proceedings of the 21st International Conference on Principles of Knowledge Representation and Reasoning, KR 2024*, ijcai.org, doi:10.24963/kr.2024/78.
  - [7] Jo Devriendt & Bart Bogaerts (2016): *BreakID: Static Symmetry Breaking for ASP (System Description)*. *CoRR* abs/1608.08447, doi:10.48550/arXiv.1608.08447.
  - [8] Jo Devriendt, Bart Bogaerts, Maurice Bruynooghe & Marc Denecker (2016): *On Local Domain Symmetry for Model Expansion*. *Theory and Practice of Logic Programming* 16(5-6), pp. 636–652, doi:10.1017/S1471068416000508.
  - [9] Christian Drescher, Oana Tifrea & Toby Walsh (2011): *Symmetry-Breaking Answer Set Solving*. *AI Communications* 24(2), pp. 177–194, doi:10.3233/AIC-2011-0495.
  - [10] Martin Gebser, Amelia Harrison, Roland Kaminski, Vladimir Lifschitz & Torsten Schaub (2015): *Abstract Gringo*. *Theory and Practice of Logic Programming* 15(4-5), pp. 449–463, doi:10.1017/S1471068415000150.
  - [11] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Patrick Lühne, Javier Romero & Torsten Schaub (2016): *Answer Set Solving With Generalized Learned Constraints*. In: *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016, OASlcs 52, Schloss Dagstuhl - Leibniz-Zentrum für Informatik*, pp. 9:1–9:15, doi:10.4230/OASlcs.ICLP.2016.9.
  - [12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2012): *Answer Set Solving in Practice*. Morgan and Claypool Publishers, doi:10.1007/978-3-031-01561-8.
  - [13] Martin Gebser, Roland Kaminski, Benjamin Kaufmann & Torsten Schaub (2019): *Multi-Shot ASP Solving With Clingo*. *Theory and Practice of Logic Programming* 19(1), pp. 27–82, doi:10.1017/S1471068418000054.
  - [14] Martin Gebser, Benjamin Kaufmann & Torsten Schaub (2012): *Conflict-Driven Answer Set Solving: From Theory to Practice*. *Artificial Intelligence* 187, pp. 52–89, doi:10.1016/j.artint.2012.04.001.
  - [15] Carla P. Gomes & Meinolf Sellmann (2004): *Streamlined Constraint Reasoning*. In: *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming, CP 2004, Lecture Notes in Computer Science 3258*, Springer, pp. 274–289, doi:10.1007/978-3-540-30201-8\_22.
  - [16] Matej Husár, Jirí Svancara, Philipp Obermeier, Roman Barták & Torsten Schaub (2022): *Reduction-Based Solving of Multi-Agent Pathfinding on Large Maps Using Graph Pruning*. In: *Proceedings of the 21st International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2022, IFAAMAS*, pp. 624–632, doi:10.5555/3535850.3535921.
  - [17] Adam Ishay, Zhun Yang & Joohyung Lee (2023): *Leveraging Large Language Models to Generate Answer Set Programs*. In: *Proceedings of the 20th International Conference on Principles of Knowledge Representation and Reasoning, KR 2023*, ijcai.org, pp. 374–383, doi:10.24963/kr.2023/37.
  - [18] Mark Law (2023): *Conflict-Driven Inductive Logic Programming*. *Theory and Practice of Logic Programming* 23(2), pp. 387–414, doi:10.1017/S1471068422000011.
  - [19] Kevin Leo, Graeme Gange, Maria Garcia de la Banda & Mark Wallace (2024): *Automatic Core-Guided Reformulation via Constraint Explanation and Condition Learning*. In: *Proceedings of the 38th AAAI Conference on Artificial Intelligence, AAAI 2024, AAAI Press*, pp. 8065–8072, doi:10.1609/aaai.v38i8.28645.

- [20] Javier Romero, Torsten Schaub & Klaus Strauch (2025): *On the Generalization of Learned Constraints for ASP Solving in Temporal Domains*. *Theory and Practice of Logic Programming* 25(2), pp. 197–224, doi:10.1017/s1471068424000462.
- [21] Karem A. Sakallah (2021): *Symmetry and Satisfiability*. In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* 336, IOS Press, pp. 509–570, doi:10.3233/FAIA200996.
- [22] Manuel A. Borroto Santana, Irfan Kareem & Francesco Ricca (2024): *Towards Automatic Composition of ASP Programs from Natural Language Specifications*. In: *Proceedings of the 33rd International Joint Conference on Artificial Intelligence, IJCAI 2024*, ijcai.org, pp. 6198–6206, doi:10.24963/ijcai.2024/685.
- [23] Arseny Skryagin, Daniel Ochs, Phillip Deibert, Simon Kohaut, Devendra Singh Dhami & Kristian Kersting (2024): *Answer Set Networks: Casting Answer Set Programming into Deep Learning*. *CoRR abs/2412.14814*, doi:10.48550/arXiv.2412.14814.
- [24] Patrick Spracklen, Nguyen Dang, Özgür Akgün & Ian Miguel (2023): *Automated Streamliner Portfolios for Constraint Satisfaction Problems*. *Artificial Intelligence* 319, p. 103915, doi:10.1016/j.artint.2023.103915.
- [25] Stefan Szeider (2025): *ASP-Bench: Problems, Ground Truths, and Solutions*, doi:10.5281/zenodo.18062939.
- [26] Stefan Szeider (2025): *Bridging Language Models and Symbolic Solvers via the Model Context Protocol*. In: *Proceedings of the 28th International Conference on Theory and Applications of Satisfiability Testing, SAT 2025, LIPIcs* 341, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 30:1–30:12, doi:10.4230/LIPIcs.SAT.2025.30.
- [27] Alice Tarzariol, Martin Gebser & Konstantin Schekotihin (2022): *Lifting Symmetry Breaking Constraints With Inductive Logic Programming*. *Machine Learning* 111(4), pp. 1303–1326, doi:10.1007/s10994-022-06146-3.
- [28] Alice Tarzariol, Martin Gebser, Konstantin Schekotihin & Mark Law (2023): *Learning to Break Symmetries for Efficient Optimization in Answer Set Programming*. In: *Proceedings of the 37th AAAI Conference on Artificial Intelligence, AAAI 2023*, AAAI Press, pp. 6541–6549, doi:10.1609/aaai.v37i5.25804.
- [29] Alice Tarzariol, Konstantin Schekotihin, Martin Gebser & Mark Law (2022): *Efficient Lifting of Symmetry Breaking Constraints for Complex Combinatorial Problems*. *Theory and Practice of Logic Programming* 22(4), pp. 606–622, doi:10.1017/S1471068422000151.
- [30] Richard Taupe, Antonius Weinzierl & Gerhard Friedrich (2020): *Conflict Generalisation in ASP: Learning Correct and Effective Non-Ground Constraints*. *Theory and Practice of Logic Programming* 20(5), pp. 799–814, doi:10.1017/S1471068420000368.
- [31] Erich Christian Teppan, Gerhard Friedrich & Georg Gottlob (2016): *Tractability Frontiers of the Partner Units Configuration Problem*. *Journal of Computer and System Sciences* 82(5), pp. 739–755, doi:10.1016/j.jcss.2015.12.004.
- [32] Florentina Voboril, Martin Gebser, Stefan Szeider & Alice Tarzariol (2026): *Code and Instances for the Paper: Streamliners for Answer Set Programming*, doi:10.5281/zenodo.18378760.
- [33] Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy & Stefan Szeider (2025): *Balancing Latin Rectangles With LLM-Generated Streamliners*. In: *Proceedings of the 31st International Conference on Principles and Practice of Constraint Programming, CP 2025, LIPIcs* 340, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, pp. 36:1–36:17, doi:10.4230/LIPIcs.CP.2025.36.
- [34] Florentina Voboril, Vaidyanathan Peruvemba Ramaswamy & Stefan Szeider (2025): *Generating Streamlining Constraints With Large Language Models*. *Journal of Artificial Intelligence Research* 84, pp. 16:1–16:20, doi:10.1613/jair.1.18965.
- [35] Toby Walsh (2012): *Symmetry Breaking Constraints: Recent Results*. In: *Proceedings of the 26th AAAI Conference on Artificial Intelligence, AAAI 2012*, AAAI Press, doi:10.1609/AAAI.V26I1.8437.
- [36] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Richter, Fei Xia, Ed Chi, Quoc V. Le & Denny Zhou (2022): *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*. In: *Proceedings of the 36th International Conference on Neural Information Processing Systems, NIPS 2022*, Curran Associates Inc., pp. 24824–24837.

- [37] Antonius Weinzierl (2013): *Learning Non-Ground Rules for Answer-Set Solving*. In: *Proceedings of the 2nd Workshop on Grounding and Transformations for Theories With Variables, GTTV 2013*.
- [38] Zhun Yang, Adam Ishay & Joohyung Lee (2020): *NeurASP: Embracing Neural Networks into Answer Set Programming*. In: *Proceedings of the 29th International Joint Conference on Artificial Intelligence, IJCAI 2020*, [ijcai.org](http://ijcai.org), pp. 1755–1762, doi:10.24963/ijcai.2020/243.
- [39] Anssi Yli-Jyrä, Masood Feyzbakhsh Rankooh & Tomi Janhunen (2023): *Pruning Redundancy in Answer Set Optimization Applied to Preventive Maintenance Scheduling*. In: *Proceedings of the 25th International Symposium on Practical Aspects of Declarative Languages, PADL 2023, Lecture Notes in Computer Science 13880*, Springer, pp. 279–294, doi:10.1007/978-3-031-24841-2\_18.

## A Encodings

This section illustrates the encodings used for the three domains studied in the paper. In particular, it provides an intuition for the meaning of the predicates appearing in the streamliners returned by our approach.

**Partner Units Problem (PUP).** The baseline PUP encoding is derived from Aschinger et al. [1] and is displayed in Listing 4. A PUP instance provides atoms of the form `zone2sensor/2`, representing the bipartite input graph, and atoms of the form `comUnit/1` providing the IDs of the (communication) units. Moreover, the constants *UCAP* and *IUCAP* are defined via the atom `maxElements(M)` with  $M = UCAP$  and `maxPU(M)` with  $M = IUCAP$ . The encoding then defines the assignments of units to zones via `unit2zone/2` and units to sensors via `unit2sensors/2`, as well as the condition for two units to be adjacent with atoms of the form `partnerunits/2`. Asserting respective constraints guarantees that the problem conditions hold.

**Sokoban.** The baseline encoding for Sokoban is shown in Listing 5. An instance provides atoms of the form `square/2`, `initial_box/2`, and `target_square/2`, representing the grid locations, the initial locations of the boxes, or their goal locations, respectively. Moreover, the atoms `initial_at(X1,Y1)` and `steps(T)` specify the initial location  $(X1, Y1)$  of the sokoban and the maximum number  $T$  of actions. The Sokoban encoding provides the output predicate `push/7`, where `push(X1,Y1,X2,Y2,DX,DY,T)` means that, at time  $T$ , the sokoban pushes a box placed in the location  $(X1, Y1)$  to the new location  $(X2, Y2)$  in direction  $(DX, DY)$  (e.g.,  $(1,0)$  means rightwards,  $(0,1)$  means upwards, etc.). Auxiliary predicates (e.g., `next/6`, `from/5`, `reach/3`, etc.) define relations used in constraints to produce a plan that complies with the rules of the Sokoban problem. Among them, `box(X,Y,T)` means that, at time  $T$ , a box is placed in the location  $(X, Y)$ .

**Towers of Hanoi.** The baseline encoding for Towers of Hanoi is given in Listing 6. An instance provides atoms of the form `disk/1`, `peg/1`, `init_on/2`, and `goal_on/2`, representing the identifiers of disks or pegs, respectively, as well as the initial and goal pegs for the disks. Moreover, the atom `moves(T)` specifies the maximum number  $T$  of moves. The encoding defines the output predicate `move/3`, where `move(D,P,T)` means that, at time  $T$ , the disk  $D$  is moved to peg  $P$ . The auxiliary predicates `time/1`, `moved/2`, `on/3`, and `blocked/3` provide available times, moved disks, the disks' pegs, and disks blocked by some smaller disk on the same peg. Respective constraints make sure that only the topmost disk of a peg is moved, it is also the smallest disk on its new peg, and the goal pegs are reached at the end of the plan.

```

zone(Z) :- zone2sensor(Z,D).
sensor(D) :- zone2sensor(Z,D).

1 { unit2zone(U,Z) : comUnit(U) } 1 :- zone(Z).
1 { unit2sensor(U,S) : comUnit(U) } 1 :- sensor(S).
:- comUnit(U), maxElements(M), M+1 { unit2zone(U,Z): zone(Z) }.
:- comUnit(U), maxElements(M), M+1 { unit2sensor(U,S): sensor(S) }.
partnerunits(U,P) :- unit2zone(U,Z), zone2sensor(Z,S), unit2sensor(P,S), U!=P.
partnerunits(U,P) :- partnerunits(P,U), comUnit(U), comUnit(P).
:- comUnit(U), maxPU(M), M+1 { partnerunits(U,P): comUnit(P) }.

```

Listing 4: Baseline Encoding for PUP

```

diff(0,1;1,0;0,-1;-1,0).

next(X1,Y1,X2,Y2,DX,DY) :- square(X1,Y1), diff(DX,DY),
                             square(X2,Y2), X2 = X1+DX, Y2 = Y1+DY.
next(X2,Y2,DX,DY)         :- next(X1,Y1,X2,Y2,DX,DY).

path(X1,Y1,X2,Y2,DX,DY) :- next(X1,Y1,X2,Y2,DX,DY), next(X1,Y1,DX,DY).
path(X1,Y1,X2,Y2,DX,DY) :- path(X1,Y1,X3,Y3,DX,DY), next(X3,Y3,X2,Y2,DX,DY).

step(1..T) :- steps(T).

box(X,Y,0) :- initial_box(X,Y).

{push(X1,Y1,X2,Y2,DX,DY,T) : path(X1,Y1,X2,Y2,DX,DY)} 1 :- step(T).

from(X1,Y1,DX,DY,T) :- push(X1,Y1,X2,Y2,DX,DY,T).
from(X1,Y1,T)         :- from(X1,Y1,DX,DY,T).

:- from(X,Y,T), not box(X,Y,T-1).

todo(T) :- target_square(X,Y), step(T), not box(X,Y,T-1).

:- todo(T), not from(X,Y,T) : square(X,Y).

box(X2,Y2,T) :- push(X1,Y1,X2,Y2,DX,DY,T).
box(X1,Y1,T) :- box(X1,Y1,T-1), step(T), not from(X1,Y1,T).

:- target_square(X,Y), steps(T), not box(X,Y,T).

free(X2,Y2,DX,DY,T) :- next(X1,Y1,X2,Y2,DX,DY), from(X1,Y1,DX,DY,T).
free(X2,Y2,DX,DY,T) :- next(X1,Y1,X2,Y2,DX,DY), free(X1,Y1,DX,DY,T),
                        not box(X1,Y1,T).

:- free(X,Y,DX,DY,T), box(X,Y,T-1).

reach(X1,Y1,1) :- initial_at(X1,Y1), step(1).
reach(X1,Y1,T) :- from(X1,Y1,T-1), step(T).
reach(X2,Y2,T) :- next(X1,Y1,X2,Y2,DX,DY), reach(X1,Y1,T), not box(X2,Y2,T-1).

:- next(X1,Y1,X2,Y2,DX,DY), from(X2,Y2,DX,DY,T), not reach(X1,Y1,T).

boxes(N) :- N = #count{X,Y : initial_box(X,Y)}.

:- step(T), boxes(N), #count{X,Y : box(X,Y,T)} != N.

```

Listing 5: Baseline Encoding for Sokoban

## B Manual Revision of Streamliners

Tables 5, 6, and 7 contrast the streamliners for PUP, Sokoban, or Towers of Hanoi, respectively, with our manual reformulations that are experimentally compared in Section 4.4. The primary objective of these reformulations is to reduce the grounding size, e.g., by turning constraint P2 / P5 over pairs  $u_1 > u_2$  of

```

time(1..T) :- moves(T).
on(D,P,0) :- init_on(D,P).

{move(D,P,T) : disk(D), peg(P)} = 1 :- time(T).

moved(D,T) :- move(D,P,T).

blocked(D,P,T) :- on(D,P,T-1), time(T).
blocked(D,P,T) :- blocked(D+1,P,T), disk(D).

:- move(D,P,T), blocked(D,P,T).
:- moved(D,T), on(D,P,T-1), blocked(D+1,P,T).

on(D,P,T) :- move(D,P,T).
on(D,P,T) :- on(D,P,T-1), time(T), not moved(D,T).

:- goal_on(D,P), moves(T), not on(D,P,T).

```

Listing 6: Baseline Encoding for Towers of Hanoi

units into P2r / P5r, where a #count aggregate lists each unit *u* only once (cf. Table 5).<sup>3</sup> For each sensor *S*, this reformulation yields linear rather than quadratic grounding size in terms of the number of units. In some cases, our reformulations also strengthen the original streamliners by dropping unnecessary side conditions. For example, the reformulation S3r of constraint S3 expresses that the target of a push action in Sokoban always contains a box in the next state, while S3 unnecessarily takes another push action into account as well (cf. Table 6). Similarly, the reformulations T2r, T4r / T6r, and T5r eliminate redundancies in constraints T2, T4, T5, and T6 for Towers of Hanoi, where the a priori different streamliners T4 and T6 are condensed to the same revised constraint T4r / T6r (cf. Table 7). We do not reformulate constraints P3, P4, and S4 as neither the grounding size nor redundancies can be cut down for them, and T3 is not listed in Table 7 because it is a comment that was produced by the LLM.

<sup>3</sup>The streamliners P2 and P5, generated in separate runs for PUP, are identical and thus have the same reformulation.

Table 5: Manually revised versions of streamliners for PUP

| Constraint | (Re)formulation                                                                                                                                         |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------|
| P1         | <code>:- comUnit(U), #count{Z: unit2zone(U,Z)} = 0,<br/>#count{S: unit2sensor(U,S)} &gt; 0.</code>                                                      |
| P1r        | <code>:- unit2sensor(U,_), not unit2zone(U,_).</code>                                                                                                   |
| P2 / P5    | <code>:- unit2sensor(U1,S), unit2sensor(U2,S), U1 &gt; U2.</code>                                                                                       |
| P2r / P5r  | <code>:- sensor(S), #count{U: unit2sensor(U,S)} &gt; 1.</code>                                                                                          |
| P3         | <code>:- unit2zone(U+1,_), not unit2zone(U,_), comUnit(U), comUnit(U+1).</code>                                                                         |
| P4         | <code>:- comUnit(U), unit2zone(U,Z1), unit2zone(U,Z2), Z1 &lt; Z2,<br/>#count{S: zone2sensor(Z1,S), zone2sensor(Z2,S)} = 0.</code>                      |
| P6         | <code>:- comUnit(U1), comUnit(U2), U1 &lt; U2, sensor(S),<br/>S = #min{S': sensor(S')}, unit2sensor(U2,S),<br/>#count{Z: unit2zone(U1,Z)} = 0.</code>   |
| P6r        | <code>mark(U):- comUnit(U), unit2sensor(U+1,S), S = #min{X: sensor(X)}.<br/>mark(U):- comUnit(U), mark(U+1).<br/>:- mark(U), not unit2zone(U,_).</code> |

Table 6: Manually revised versions of streamliners for Sokoban

| Constraint | (Re)formulation                                                                                                                     |
|------------|-------------------------------------------------------------------------------------------------------------------------------------|
| S1         | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T), X1 &lt; X3.</code>                                                   |
| S1r        | <code>:- diff(DX,DY), step(T), #count{X1 : push(X1,Y1,X2,Y2,DX,DY,T)} &gt; 1.</code>                                                |
| S2         | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T),<br/>X1 = X3, Y1 &lt; Y3.</code>                                      |
| S2r        | <code>:- diff(DX,DY), step(T), square(X,Y),<br/>#count{Y1 : push(X,Y1,X2,Y2,DX,DY,T)} &gt; 1.</code>                                |
| S3         | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), push(X3,Y3,X4,Y4,DX,DY,T+1),<br/>X1+Y1 &gt; X3+Y3, not box(X2,Y2,T), X2 != X3, Y2 != Y3.</code> |
| S3r        | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), not box(X2,Y2,T).</code>                                                                        |
| S4         | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), push(X2,Y2,X1,Y1,ODX,ODY,T+1).</code>                                                           |
| S5         | <code>:- push(X1,Y1,X2,Y2,DX,DY,T), push(X1,Y1,X3,Y3,DX,DY,T),<br/>X2 = X3, Y2 &lt; Y3.</code>                                      |
| S5r        | <code>:- path(X1,Y1,X1,Y,DX,DY), step(T),<br/>#count{Y2 : push(X1,Y1,X1,Y2,DX,DY,T)} &gt; 1.</code>                                 |
| S6         | <code>:- push(X1,Y1,X2,Y2,1,0,T), push(X3,Y3,X4,Y4,-1,0,T),<br/>X2 = X4, Y2 &lt; Y4.</code>                                         |
| S6r        | <code>stop(X2,Y2,DX,0,T):- push(X1,Y1,X2,Y2,DX,0,T).<br/>:- stop(X,Y1,1,0,T), stop(X,Y2,-1,0,T), Y1 &lt; Y2.</code>                 |

Table 7: Manually revised versions of streamliners for Towers of Hanoi

| Constraint | (Re)formulation                                                                                                                                                                                                                                                                 |
|------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| T1         | <code>:- moved(D1,T-1), moved(D2,T), D2 &lt;= D1, disk(D1+1).</code><br><code>mark(D,T):- moved(D,T-1), time(T), disk(D+1).</code>                                                                                                                                              |
| T1r        | <code>mark(D,T):- mark(D+1,T), disk(D).</code><br><code>:- mark(D,T), moved(D,T).</code>                                                                                                                                                                                        |
| T2         | <code>stable(1,P,T):- on(1,P,T), goal_on(1,P), time(T).</code><br><code>stable(D,P,T):- on(D,P,T), goal_on(D,P), stable(D-1,P,T),</code><br><code>                  disk(D), disk(D-1), time(T).</code><br><code>:- move(D,P,T), stable(D,P_prev,T-1), on(D,P_prev,T-1).</code> |
| T2r        | <code>stable(1,P,T):- on(1,P,T-1), goal_on(1,P).</code><br><code>stable(D,P,T):- on(D,P,T-1), goal_on(D,P), stable(D-1,P,T).</code><br><code>:- moved(D,T), stable(D,P,T).</code>                                                                                               |
| T4         | <code>:- disk(D), move(D,P1,T1), move(D,P2,T2), move(D,P1,T3),</code><br><code>      T1 &lt; T2, T2 &lt; T3, T3 = T2 + 1, P1 = P2.</code>                                                                                                                                       |
| T4r / T6r  | <code>:- move(D,P,T), move(D,P,T+1).</code>                                                                                                                                                                                                                                     |
| T5         | <code>:- move(D1,P1,T), move(D2,P2,T+1), D1 &lt; D2,</code><br><code>      on(D1,Px,T-1), on(D2,Py,T), Px = Py, P1 != Py, P2 = Px.</code>                                                                                                                                       |
| T5r        | <code>:- move(D,P,T), on(D,P,T-1).</code>                                                                                                                                                                                                                                       |
| T6         | <code>:- time(T), T &gt; 2, move(D,P1,T-1), move(D,P2,T-2), P1 = P2,</code><br><code>      move(D,P1,T).</code>                                                                                                                                                                 |