arXiv:2602.01171v1 [cs.AI] 1 Feb 2026

ASP-Bench：从自然语言到逻辑程序*

Stefan Szeider

维也纳工业大学算法与复杂性组，奥地利维也纳
www.ac.tuwien.ac.at/people/szeider/

摘要

将自然语言规范自动翻译为逻辑程序是一项具有挑战性的任务，会影响神经符号工程。我们提出了 ASP-Bench，这是一个基准，包含 128 个自然语言问题实例，即 64 个具有简单和困难变体的基础问题。它用于评估将自然语言问题翻译为答案集程序（ASP）的系统；答案集程序是逻辑编程的一种重要形式。该基准系统性地覆盖了 ASP 的各种特性，包括选择规则、聚合和优化。每个问题都包含参考验证器，用于检查解是否满足问题规范。

我们从七个大体上相互独立的推理方面刻画这些问题（优化、时间推理、默认逻辑、资源分配、递归、空间推理和定量复杂度），从多维角度展现建模难度。

我们采用基于 ReAct（推理与行动）框架的智能体方法对该基准进行测试，该方法实现了完全饱和，表明结合求解器反馈的反馈驱动迭代细化为用答案集编程对自然语言进行建模提供了一种可靠且稳健的方法。我们对多次智能体运行的分析使我们能够深入了解决定问题建模难度的因素。

1 引言

答案集编程（ASP）是一种声明式编程方法，能够刻画知识表示与推理中的许多问题 [8]。问题被编码为逻辑程序，其答案集对应于解。特别是，答案集编程的声明式特性支持模块化开发、紧凑的问题表示以及可证明正确的系统设计。该范式受益于高效的通用求解器，例如 clasp [6], clingo [7], 和 DLV [13], ，它们将问题建模与求解过程分离开来。

在答案集编程中，问题被表述为规则和约束，而不是逐步的算法。给定一个逻辑程序，求解器会自动计算解。这类似于编写 SQL 查询而非编写过程式代码：你描述解必须满足什么，而不是如何找到它。

将自然语言（NL）规格说明转换为逻辑程序，仍是答案集编程得到更广泛应用的一大障碍，因此推动了自动化这一任务的研究。早期方法依赖受控自然语言（CNL），通过将输入限制在形式化定义的语法范围内，确保翻译可靠 [3]。近年来，大语言模型的出现带来了更灵活的方法。典型例子包括多阶段流水线，将自然语言转换为受控自然语言表示 [15], ；使用通用大语言模型直接进行一次性生成 [11], ；以及像 LLASP 这样的经过微调的领域专用模型 [4]。然而，实验

*即将发表于第二届神经符号软件工程国际研讨会（NSE@ICSE 2026）论文集。由奥地利科学基金会（FWF）项目 10.55776/P36688 和 10.55776/COE12 资助。

评估表明，一次性生成往往不够可靠；Borroto等人 [15] 发现，ChatGPT-3.5 为 21 个图问题中的 17 个生成了有缺陷的程序。这表明需要一种更稳健的智能体式创建过程，让智能体利用求解器反馈，在多个推理-行动循环中不断完善和调试程序。

其他神经符号系统（如NeurASP [20], ）假定人类专家已经编写了正确的逻辑程序。因此，神经网络仅提供感知输入（例如识别图像中的数字），并将其输入预先存在的规则。可微求解器将答案集编程计算转化为图神经网络以实现GPU并行 [16] ，但要求程序本身必须手动编码。

NL2ASP翻译方面的近期工作产生了NL-ASP-200基准 [11], ，其中包含200个问题，已报告的最高准确率为58.5% [15]。然而，这些基准存在关键局限。首先，它们使用精确字符串匹配进行评估。Ishay等人承认，该方法会拒绝四分之三在语义上正确、仅在句法上与参考解不同的程序。其次，这些问题是从在线论坛和教科书中机会性地抽样得到的，对答案集编程语言构造的系统覆盖有限。

相关形式体系的基准——一阶逻辑 [9], 时序逻辑 [5], 命题逻辑 [2], 以及约束规划 [14]——表明，使用自动化求解器进行语义验证对于可靠评估至关重要。

我们提出了**ASP-Bench**，这是一个旨在解决上述局限的基准。ASP-Bench 包含 128 个问题实例（64 个基础问题，每个问题都有简单和困难两个版本），系统地覆盖了核心 ASP 构造，范围从基本规则到选择规则、聚合和优化。每个问题都配有参考验证器（“真实值”），通过检查解是否满足问题规范来进行语义验证，而无需要求字符串完全匹配。

为刻画该基准的多样性，我们分析了 64 个困难问题变体，以识别导致建模困难的反复出现的推理方面。每个问题包含 1 至 6 个此类方面，从而能够分析哪些方面的组合会使 ASP 建模更加困难。

为展示该基准并为未来的比较建立基线，我们采用了一种基于 ReAct（推理与行动）框架的智能体方法 [21], ，该方法通过求解器反馈进行迭代细化。我们的系统由一个单提示自主大语言模型智能体构成，它负责所有开发和测试任务，不遵循固定的工作流程或预定义的流程。因此，与一次性生成方法 [11] 或静态组合流程 [15], 不同，该智能体会根据求解器反馈动态决定采取的行动，利用 clingo 的错误消息和未通过的测试用例，在多个推理-行动循环中迭代修复和改进 ASP 程序。该方法在 ASP-Bench 的简单和困难问题上均实现了完全饱和，展示了反馈驱动的迭代细化在 ASP 程序合成中的有效性。我们还比较了两种基于 MCP 的交互式 ASP 建模方法（其中一种与我们的智能体共享求解器集成），发现通用 Python 执行优于专用声明式接口。

2 示例：骑士与无赖

我们给出一个自然语言问题及其逻辑程序译文的示例。该示例基于问题 03（骑士与无赖），这是 ASP-Bench 的简单问题变体之一。如果一个人是骑士，那么他总是说真话；如果他是无赖，那么他总是说谎。已知 Alice 说“Bob 是无赖”，Bob 说“Alice 和 Charlie 属于同一类型”，而 Charlie 说“Alice 是骑士”，任务是确定每个人的类型。

建模过程首先通过 clingo 库生成 Python 代码来定义：

```
自clingo 导入 Control ctl  =
Control ()
```

```
ctl . add (" base ", [], """ 人 ( Alice ;␣ Bob ;␣
Charlie ). {␣ 骑士 (X)␣:␣ 人 (X)␣}. 无赖 (X)␣:-
␣ 人 (X),␣ not ␣ 骑士 (X ).

    ... """) ctl . ground ([( " base ", [])]) ctl .
solve ( on_model =extract_solution )
```

这个 Python 程序生成了以下逻辑程序。

```
person(alice; bob; charlie). { 骑士(X) :
person(X) }. 无赖(X):- person(X), not骑士(
X).

:-骑士(alice)，骑士(bob)。 %:-无赖
(alice)，无赖(bob)。

:- knight(bob), knight(alice), knave(charlie).
:- knight(bob), knave(alice), knight(charlie).
:- knave(bob), knight(alice), knight(charlie).
:- knave(bob), knave(alice), knave(charlie).  %

:-骑士(charlie), 无赖(alice). %:-无赖
(charlie), 骑士(alice).
```

该编码的关键要素包括用于生成候选项的选择规则、用于定义互补谓词的完备性公理，以及用于确保一致性的完整性约束。每条陈述都会产生两个约束（由骑士说出时为真，由无赖说出时为假）。

3 ASP-Bench 基准测试

3.1 问题构建流程

我们借助编码智能体生成了 ASP-Bench 中的 64 个基础问题。整个过程分为以下几个阶段：构思问题、起草自然语言的问题描述、创建 ASP 编码，以及测试解答。编码智能体与人工监督在这一过程中反复协作。通过这种反复交流，我们构建了 64 个简单的问题变体，涵盖了多种推理模式，包括逻辑谜题、图问题、调度、优化和约束满足。其中既有全新的问题，也有对已知问题的变体。我们使用 Claude Sonnet 4.5 作为编码智能体 Claude Code 的主要大语言模型，并通过 consult7 工具借助 Gemini Pro 2.5（推理模式）提供咨询支持[1], 。该工具通过模型上下文协议（MCP）并行查询多个前沿模型，提供结构化代码分析和推理能力[2]。这为复杂的设计决策提供了宝贵的参考。

在建立了 64 个简单问题及其验证器后，我们通过系统性地增加问题复杂度来构建困难变体。这种增强既包括定量方面（更大的实例、更多的约束），也包括定性方面（额外的约束类型、多目标优化、更复杂的推理模式）。

对于许多困难问题，我们使用了预设解：先生成一个已知的可行解，然后围绕它构建约束。事实证明，这种方法比无约束

[1]https://pypi.org/project/consult7/ [2] https://modelcontextprotocol.io/

问题生成更可靠，因为验证约束是否容许解在计算上比求解任意约束系统更容易。因此，我们可以保证困难问题是适定的且存在解，这对基准的可靠性与可复现性至关重要。我们通过多轮智能体-人工交互迭代，确保问题在合理的时间范围内仍可求解（求解器运行时间为$\leq$20 秒）。

3.2 语义验证

ASP-Bench采用语义验证，而非语法字符串匹配。每个问题都有一个真实值，其形式为参考验证器：一个用于检查所提解决方案是否符合问题规范的Python脚本。验证器从标准输入加载JSON解决方案，检查是否满足所有问题约束，并按以下格式输出验证结果

每个问题描述都会明确说明所需的JSON字段及其语义，确保解决方案格式要求明确无歧义，且正确解决方案因格式错误而无法通过验证的概率极低。

参考验证器会接受任何正确满足约束的解决方案。这一点至关重要，因为问题往往有许多有效解，因此仅与某个固定解进行比较是不够的。

对于优化问题，我们还关注解决方案的最优性。参考验证器会检查智能体的解决方案是否在满足所有约束的同时达到已知最优值。

需要指出的是，验证是在自然语言到ASP的构造完成后进行的。因此，在ASP-Bench上接受测试的任何方法都必须依靠自身的测试和验证机制，因为它无法访问真实值。

3.3 问题特征分析

我们从七个方面分析 ASP-Bench 问题，这些方面代表了反复出现的推理模式：

1. OPT：使用最小化/最大化语句的优化问题；2. TEMP：需要顺序约束的时序或序列推理问题；3. DEFAULT：包含软约束或偏好的问题；4. RESOURCE：具有容量限制的资源分配问题；5. RECURSIVE：需要环检测或不动点计算的推理问题；6. SPATIAL：基于网格或邻域逻辑的问题；7. QUANT：定量复杂度较高的问题（包含七个或更多不同约束）。

这些方面大体上相互独立，因此一个问题可以同时包含多个方面。

方面分类体系是在完成问题集后通过事后分析得出的，它识别了导致答案集编程建模难度的常见模式。图1展示了七个方面在64个困难问题变体中的分布（按下文所述的“难度”排序）。

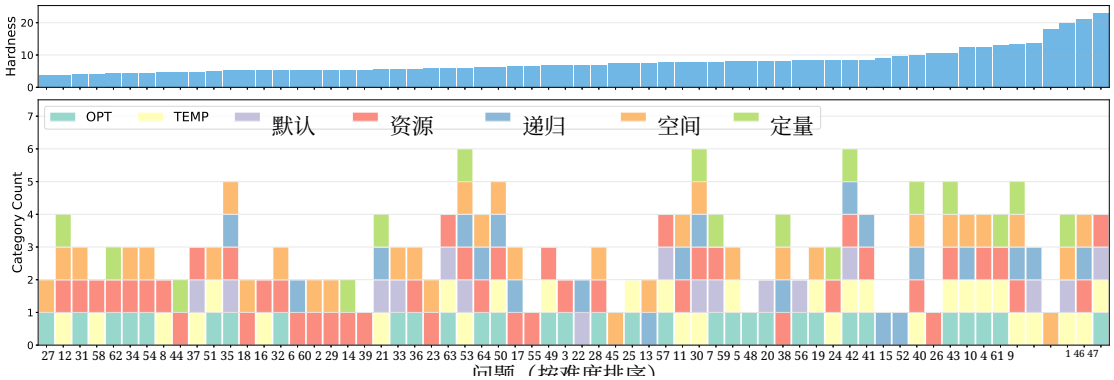


图1：顶部：每个问题的平均 python_exec 调用次数（难度衡量指标）。底部：每个问题涉及的推理方面。问题按难度从低到高排序。

4 ASP智能体

ASP-Agent 是我们在约束建模领域中介绍的通用编码智能体架构的一种改编版本 [19]。其实现已作为开源软件提供[3]。该智能体基于 ReAct（推理与行动）框架 [21], , 使大语言模型能够在推理、工具执行和观察的迭代循环中运行。该架构的一个重要组成部分是持久化的 IPython 内核，它能在多次代码执行之间保留状态，使智能体能够逐步构建、测试和调试解决方案。这种设计清晰地将与领域无关的代码创建和执行能力，与特定任务的专业知识分离开来；后者完全通过项目提示注入，而非硬编码到智能体逻辑中。

该智能体使用两种工具：python_exec 和 save_code。反复调用 python_exec 推动了大部分迭代开发。python_exec 的调用次数可作为问题难度的替代指标，反映智能体为得出解答所需的推理-行动循环次数。save_code 工具仅在最后调用一次，用于表明编码已完成，并将最终代码写入磁盘。运行代码后会生成 JSON 格式的解答，供参考验证器检查。

智能体中只有两个组件专门针对答案集编程：（1）全面的项目提示 clingo.md[4]；（2）一个用于加载 *clingo* Python 库的参数 [7], , 该库提供了建模答案集编程问题以及与 clingo ASP求解器交互所需的 API。

项目提示会指导智能体如何使用该库。此外，它还提供了一份详细指南，涵盖必须遵守的答案集编程语法规则（例如变量安全性、聚合放置）、建议的工作流程、调度和图问题的常见建模模式，以及关键反模式清单（例如选择规则中基数设置错误、低效的生成并过滤逻辑）。

项目提示分为五个主要部分：
(1)任务简报与核心要求：确立总体目标和不可妥协的约束，

(2) 关键交战规则：详细说明必须遵循的答案集编程语法、封闭世界假设等逻辑原则，以及必需的五步工作流（规划、分析与建模、实现、求解与提取、格式化与验证），

[3]https://github.com/szeider/agentic-python-coder [4]提示可在此处获取：
https://github.com/szeider/agentic-python-coder/blob/main/examples/clingo/ clingo.md

(3) *Clingo API* 与实现指南：提供针对 Python 的实现指导，以及适用于时序问题的三步动作建模模式，

(4) 问题求解模式库：介绍适用于常见问题类型的可复用答案集编程模式，包括指派、图、调度和时序推理问题，以及

(5) 调试与高级技术：介绍性能优化策略，例如尽早限制选择并避免基化爆炸。这种结构化设计为智能体提供了基础知识和实用指导，帮助其逐步构建和调试答案集编程模型。

我们通过迭代细化过程开发了 ASP 项目提示词。我们在 ASP-Bench 中选取了几个简单问题，测试了一个包含基本 API 指令的初始提示词。我们系统地分析了智能体执行日志中的失败情况，例如语法错误、不安全变量声明以及聚合使用不当。我们据此更新了提示词，添加明确的规则、最佳实践和具体示例，以解决这些问题。不过，为了使提示词能够泛化到未见过的问题，我们避免针对个别问题过拟合。我们重复这一过程进行了数轮迭代。随后，我们在全部简单和困难问题变体上评估了最终提示词。我们观察到学习速度快得出乎意料，因此在五轮迭代后停止了这一过程。

5 实验评估

5.1 实验设置

我们使用 Claude Sonnet 4.5（2025年9月29日版本）作为底层语言模型，通过 OpenRouter 在每个问题上运行 ASP-Agent，并将温度设为 0.0、最大令牌数设为 16384，同时启用流式输出。困难问题的智能体编码过程超时限为 1200 秒，简单问题则为 600 秒。每次求解器调用的时限为 20 秒。所有运行均在超时限内顺利完成。我们对每个困难问题进行了三次独立运行，以评估一致性；对简单问题进行了两次运行。

ASP-Bench 基准测试集以及全部 128 个问题描述、真实值验证器和示例解决方案均可在 Zenodo 获取 [17]。我们以加密档案的形式提供数据（密码：qB5xjsoFtdSZWTm），以防止未来实验中的训练集受到污染。

5.2 简单问题和困难问题上的饱和

ASP-Agent 正确解决了所有实例，包括简单变体上的两次运行和困难变体上的全部三次运行。

我们报告了每个困难问题三次运行中 python_exec 调用次数的平均值：平均为 7.7 次。我们观察到问题难度存在显著差异。最简单的问题平均需要 3.7 次 python_exec 调用，最难的问题则需要 26.0 次。最难的问题依次为问题 47（DNA序列组装；26.0 次）、问题 46（银河恶魔城生成；21.0 次）、问题 01（谁是凶手；20.0 次）、问题 09（数织求解器；17.2 次）和问题 61（历史反事实；13.7 次）。最简单的问题依次为问题27（皇后支配；3.7 次）、问题 12（斑马谜题；3.7 次）、问题 31 （网络流；4.0 次）和问题 58 （考试排程；4.0 次）。

完整结果列表见表 1，其中执行次数和令牌使用量均为多次运行的平均值。

5.3 问题难度与推理方面

为了刻画哪些因素会使问题对智能体而言更困难，我们考察了第3.3节所定义的每个问题所涉及的推理方面。结果如图1所示。

表 1：ASP-Bench 结果：执行调用和令牌使用量（×1000，输入 I=、输出 O=），为多次运行的平均值（简单问题运行 2 次，困难问题运行 3 次）。

	Easy			Hard				Easy			Hard		
# 问题	Ex	I	O	Ex	I	O	# 问题	Ex	I	O	Ex	I	O
01 谁是凶手	4.5	154	4	20.0	622	23	33 独立集	3.5	126	3	5.7	154	7
02 加权图着色	3.0	119	3	5.3	155	6	34 支配集	4.0	136	3	4.3	116	5
03 骑士与无赖	2.0	96	2	7.4	202	10	35 反馈顶点集	5.5	174	4	5.3	141	6
04 积木世界	4.5	163	4	10.8	314	11	36 拉丁方	2.0	99	3	6.0	149	5
05 电路诊断	13.5	588	23	9.2	355	18	37 汽车排序	3.5	133	4	4.7	126	5
06 医院匹配	3.0	123	5	5.0	270	7	38 蛋白质结构	4.5	165	5	8.3	208	7
07 Hamilton P Wtd	4.0	141	3	8.0	196	7	39 ByzantineGenerals	4.0	136	3	5.7	181	9
08 会议调度	4.5	175	5	4.7	130	7	40 仓库	3.0	120	4	9.0	402	19
09 数织求解器	3.5	152	7	17.2	626	20	41 论证	5.0	155	3	9.7	279	8
10 资源分配 I	4.5	160	4	10.0	324	14	42 基因调控网络	2.0	97	3	4.3	137	5
11 锦标赛排名	3.5	129	3	7.7	209	8	43 量子电路	4.5	169	5	12.3	377	19
12 斑马谜题	4.0	160	7	3.7	112	6	44 非传递骰子	3.0	137	4	4.7	135	5
13 作业车间调度	3.5	137	5	7.7	220	11	45 囚徒困境	7.5	223	6	7.3	211	7
14 密码算术	3.0	118	3	5.3	141	5	46 银河恶魔城生成	7.0	232	7	21.0	929	30
15 旅行锦标赛	10.5	339	8	10.0	321	11	47 DNA序列组装	6.0	194	5	26.0	922	23
16 护士排班	19.5	724	17	5.3	128	4	48 填字游戏	12.0	360	10	8.0	229	9
17 装箱问题	5.0	163	4	6.7	181	7	49 拍卖	2.5	110	4	7.0	189	8
18 幻方	5.0	161	3	5.3	136	6	50 元胞自动机	6.0	192	6	6.7	188	6
19 课程表	3.0	123	4	8.3	234	11	51 反弹机器人	6.5	223	9	5.0	159	9
20 集合覆盖	4.0	139	3	8.3	213	7	52 尼姆游戏	5.0	164	5	10.7	291	11
21 加权顶点覆盖	3.0	119	3	3.3	114	4	53 斯坦纳树	4.0	144	5	6.3	176	10
22 团查找	4.5	148	3	7.0	168	6	54 图划分	2.5	110	3	4.3	118	5
23 资源分配 II	3.0	121	3	6.0	160	8	55 食谱规划	6.5	213	7	7.0	195	9
24 工作流优化	5.5	205	5	8.3	201	7	56 音乐创作	4.5	150	4	8.3	225	9
25 数独扫雷	6.0	189	5	7.7	257	9	57 密室逃脱	5.0	161	4	7.7	233	11
26 汉诺塔	6.5	213	5	12.3	368	12	58 考试排程	5.5	178	5	4.0	110	5
27 皇后支配	4.5	152	3	3.7	95	3	59 策略性投票	11.0	388	12	8.0	228	10
28 图同构	5.0	166	4	7.3	218	9	60 生态系统平衡	6.0	199	7	5.3	133	4
29 逻辑网格谜题	3.5	130	4	5.3	154	8	61 历史事实	4.0	143	4	13.7	374	12
30 团队组建	4.5	149	4	7.7	211	9	62 药物相互作用	8.5	302	12	4.3	129	7
31 网络流	4.5	157	5	4.0	120	6	63 地牢生成	11.0	396	18	6.0	205	13
32 频率分配	2.5	125	4	5.3	138	5	64 社交网络影响	7.0	225	7	6.3	176	8

令人惊讶的是，问题难度（以 python_exec 调用次数衡量）与推理方面的数量并无显著相关性。执行次数从涉及两个方面的问题（最简单的一组）的 5.9 次，到涉及四个方面的问题（平均而言最困难的一组）的 10.8 次不等，而涉及单个方面的问题平均执行 9.3 次。这表明，与其他问题领域相比，某些问题领域对我们的智能体而言要困难得多，且这与推理方面的数量无关；困难源于特定领域内推理的深度和复杂性，而非仅仅源于涉及多个推理方面的广度。

让我们来看一个例子，以说明这一规律。问题9（数织求解器）仅涉及空间方面。由于需要对复杂的网格约束进行推理，它是最困难的问题之一，平均需要执行17.2次。另一方面，问题27（皇后支配）结合了空间和优化两个方面，却是最简单的问题之一，平均需要执行3.7次，因为其约束结构便于直接建模。问题47（DNA序列组装）结合了四个方面（优化、TEMP、默认和资源），是最困难的问题，平均需要执行26.0次。我们认为，它的难度源于复杂的重叠约束，而非涉及方面的广度。

5.4 智能体活动模式

为了了解问题变得更困难时智能体如何分配精力，我们对所有 3×64 次困难问题运行的智能体执行记录进行了自底向上的分析，并将每次执行尝试归类为八种不同的活动类型：

数据设置：问题界定和数据初始化；初始模型：首个 ASP模型的构建；规则添加：增量式约束添加；逻辑验证：针对特定规则和约束的定向测试；错误纠正：调试和修复错误；完整求解：完整解的尝试；逻辑优化：修改现有 ASP 规则以纠正错误；解的格式化：最终确定输出格式和呈现方式。

我们根据总执行次数将运行分为四个难度范围： ≤ 10 次执行（较简单的问题）、11–20次执行（中等难度）、21–30次执行（具有挑战性的问题）以及≥ 31 次执行（最困难的问题）。图2展示了这些范围内活动分布的分析结果。该图揭示了智能体的行为如何适应问题的复杂性。对于较简单的问题（1–10次执行），我们观察到各项活动之间的努力分配相对均衡，智能体能够从初始模型快速推进到验证和解。随着问题变得更加困难（11–20次和21–30次执行调用），逻辑验证在活动分布中所占的比例更大。这项活动消耗的执行调用总数比例不断增加。这表明，较困难的问题需要采用谨慎的增量式方法。智能体会在尝试完整解之前，先测试单个约束和规则。对于最困难的问题（$31+$ 次执行，仅由DNA序列组装和数织求解器两个运行表示），完整求解尝试的次数明显更多，因为智能体通过多个周期迭代优化完整模型。即使对于最困难的问题，错误纠正所占的活动比例也低于10%。这表明，迭代优化方法能够提供反馈，从而预防错误，而不必进行大量的事后调试。

图2右侧面板展示了另外两种模式：每次执行时间从简单问题的22秒下降到最困难问题的15秒，而输入/输出令牌比则从24:1上升到45:1。更困难的问题涉及更短、更聚焦的执行周期，在生成每次响应之前，智能体会读取比例更高的上下文。

6 智能体建模剖析

为了了解 ASP-Agent 如何在实践中应用项目提示，我们来详细考察问题26（汉诺塔）。这个问题展示了智能体在问题求解策略上的灵活性，表明不同运行过程中的执行模式存在显著差异：最短的一次运行仅完成了6次 python_exec 调用，而最长的一次则需要20次调用，相差超过3倍，揭示了针对同一问题所采用的根本不同的方法。

6.1 问题复杂度：朝圣者之旅

该问题是经典的四盘四柱汉诺塔问题的一种变体，额外要求每个圆盘从源柱 A 移动到目标柱 D 的过程中，都必须经过中间柱 B 和 C。这一“朝圣之旅”约束排除了标准的汉诺塔算法，使问题从算法执行转变为复杂的状态空间搜索。预期的最优解需要 19 次移动，为时间规划设定了颇具挑战性的搜索深度上限。该问题的形式化描述采用以下参数：状态表示（每个时间步圆盘在 4 根柱子上的位置）、动作（遵循标准汉诺塔规则的有效移动操作）、时序约束（圆盘移动轨迹必须包含对柱子 B 和 C 的强制访问），以及优化目标（在满足所有约束的前提下尽量减少总移动次数）。

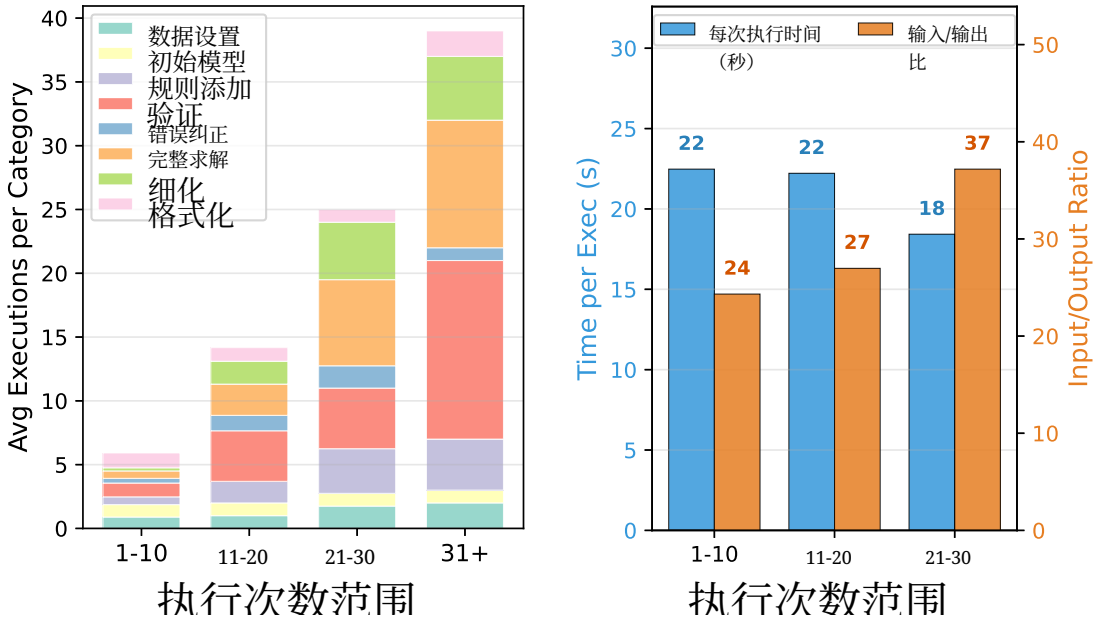


图2：左：智能体活动类型在各难度范围（1–10、11–20、21–30、31+ 次执行）上的分布。右：智能体行为指标，显示每次执行时间（蓝色，左轴）和输入/输出令牌比（橙色，右轴）。

6.2 专家策略（6次调用）

在最短的一次运行中，我们的智能体展现出专家般的表现：它立即识别出该问题具有时间规划问题的结构。它以极少的迭代，构建出完整且准确的 ASP 模型。

调用 1–2：问题分析与数据设置。 智能体阅读问题描述，提取问题参数（4 个圆盘、4 根柱子、19 步移动上限），并定义状态谓词和动作模式。

调用 3–4：构建核心答案集编程模型。 智能体以整体方式构建了一个全面的模型，定义了：(1) 表示圆盘位置的状态流 on(Disk, Peg, Time)，(2) 带有前置条件和效果的动作 move(Disk, FromPeg, ToPeg, Time)，(3) 用于描述移动之间圆盘位置持续不变的框架公理，以及 (4) 标准的汉诺塔约束，确保没有圆盘位于比它小的圆盘上方。

调用 5：朝圣者之旅约束。 随后，智能体添加了时间轨迹约束，以确保每个圆盘都访问过两个中间柱，并使用 visited/2 谓词跟踪每个圆盘在移动序列中访问过哪些柱子。

调用 6：验证与输出。 智能体运行了 clingo 求解器（不到 1 秒便找到解），验证移动序列满足所有约束，并将完整的移动序列整理为 JSON 输出。

这表明，当智能体以最高效率运行时，会呈现线性流程（分析、建模、约束、求解）。由于模型第一次尝试就正确，因此无需进一步调试。以下是智能体的输出。

顶部(D,P,T) :-在_柱子(D,P,T), 非有_更小(D,P,T). 有_更小(D,P,T) :- 在_柱子(D,P,T),在_柱子(D2,P,T), D2 < D. 1 { 移动(D,P1,P2,T) : 顶部(D,P1,T), 柱子(P2), P1!=P2 }1 :- 时间(T).

在_柱子(D,P2,T+1) :- 移动(D,_,P2,T). 在_柱子(D,P,T+1):-
在_柱子(D,P,T), 非 移动(D,P,_,T). 已访问(D,P) :- 移动(D,_,
P,_).:-圆盘(D), 非 已访问(D,b).:-圆盘(D), 非 已访问(
D,c).

6.3 调试策略（20 次调用）

最长的一次运行展现出截然不同且令人瞩目的方法，说明智能体在初次尝试失败后采用了高明的调试方法。智能体在自己的总结中明确指出：“使用 19 步的初次尝试失败了，因为求解器难以应对庞大的搜索空间。通过测试较小的实例（1–3 个圆盘），有助于验证模型正确性。”

调用 1–8：初始模型尝试及搜索空间带来的挑战。智能体构建了一个初始 ASP 模型，但求解器未能在合理的时间内找到解。模型要么过于复杂，要么存在低效之处，导致无法实际求解。智能体没有简单地反复调整同一个模型，而是转而采用诊断策略。

调用 9–14：为验证而简化。意识到性能问题后，智能体采用了一种巧妙的元认知策略：将问题从 4 个圆盘简化为 1–3 个圆盘，缩短时间范围以匹配较小的问题规模，在这些易于求解的实例上测试核心 ASP 逻辑，并验证规则是否正确编码了汉诺塔约束和状态转移。这种先在较简单版本上自我验证的方法相当高明：智能体提出了一个假设（逻辑正确，问题出在规模扩展上），并对其进行了系统测试。较小的实例均在不到 0.2 秒内成功求解，证实核心建模方法是可靠的。

调用 15–18：扩展规模并优化。 确信逻辑无误后，智能体将规模缩小至 4 个圆盘，并通过收紧中间变量的界限、添加对称性破缺约束、重新表述部分规则以提高基化效率，以及针对递增的时间范围（12、15、19 次移动）重新测试，进一步细化了模型。

调用 19–20：最终解与验证。 随后，优化后的模型在时限内求解完成。智能体验证了解轨迹，确认全部 4 个圆盘都访问过 B 和 C，并对输出进行了格式化。

这次运行展现了智能体的韧性和策略灵活性。面对失败时，智能体没有简单地重试，而是进行了诊断、简化、验证和扩展。这一结果与我们在图2中展示的活动模式研究发现一致：运行时间越长，达到解之前进行的验证循环、增量测试和策略细化就越多。以下代码清单展示了采用细化方法并使用显式前置条件的最终 ASP 编码。

```
top(D,P,S) :-on(D,P,S), not blocked(D,P,S).
blocked(D,P,S) :- on(D,P,S), on(D2,P,S),D2 < D. 1
{ move(D,P1,P2,S) : disk(D), peg(P1), peg(P2), P1!=
P2 }1 :- step(S).:-move(D,P1,_,S), not
top(D,P1,S-1).:- move(D,_,P2,S), top(D2,P2,S-1),
D > D2. on(D,P2,S) :-move(D,_,P2,S). on(D,P,S):-
on(D,P,S-1), not move(D,P,_,S). visited(D,P) :-
move(D,_,P,_).:-disk(D), not
visited(D,b).:-disk(D), not visited(D,c).
```

表 2：MCP 方法在 ASP-Bench 中 10 个最困难问题上的比较（每个问题运行 3 次）。列**p** 显示 3 次运行中的成功次数；工具调用次数和时间为平均值。

	ipython-mcp			**mcp-solver-asp**		
问题	**p**	**次数**	**时间**	**p**	**次数**	**time**
01 谁是凶手	3	25.3	277	3	61.0	290
04 积木世界	3	9.3	170	2	34.7	276
09 数织	3	20.0	207	3	51.0	276
10 资源分配	3	11.0	166	2	53.3	327
26 汉诺塔	3	19.0	300	3	44.7	338
43 量子电路	3	14.7	254	2	44.0	292
46 银河城	3	18.0	248	3	49.0	330
47 DNA序列	3	18.7	278	3	51.7	344
52 尼姆游戏	3	9.7	137	3	35.0	252
61 历史事实	3	14.0	121	3	48.0	249
总计/平均值	**30**	**16.0**	**216**	**27**	**47.2**	**297**

6.4 对比策略

执行调用次数的高方差导致资源消耗出现显著差异。专家运行耗时 101 秒，使用了 14 万个输入 token 和 4.8 千个输出 token；而调试运行耗时 574 秒，使用了 69 万个输入 token 和 1.97 万个输出 token——资源消耗约为前者的 5 倍。显然，两次运行得到的最优解相同，均为 19 次移动。

7 MCP 集成

我们的自主建模系统也可以通过将自定义 ReAct 智能体替换为其他智能体系统（例如编码智能体或聊天机器人应用），并通过模型上下文协议（MCP）连接到求解器接口来运行。该协议提供了一个标准化的开源接口，用于将语言模型与外部工具和数据源连接起来 [1,10]。MCP 于 2024年末由 Anthropic 推出，目前由 Linux 基金会托管，已成为智能体-工具交互的事实标准，并得到主要人工智能平台的采用，也集成到了开发环境中。

我们比较了两种基于 MCP 的 ASP 建模方法。

第一种是 **ipython-mcp**，即用外部智能体系统替换我们系统中的 ReAct 智能体。这台 MCP 服务器提供一个持久化的 IPython 内核，并配备用于代码执行（python_exec）、状态管理（python_reset）和会话控制的 MCP 工具。通过项目提示（clingo.md）实现领域适配；它采用与我们独立智能体架构相对应的设计，并支持完整的 Python 控制流。

第二种，**mcp-solver-asp** [18]，通过基于条目的工具（add_item、replace_item、delete_item、get_model、solve_model）提供专门的声明式模型编辑，并由 MCP-Solver 发行版中包含的指令提示引导。这确保了纯声明式工作流，让智能体操作单条 ASP 规则，而非过程式代码。

实验 我们根据表 1 中的平均执行次数，选取了 ASP-Bench 中最困难的 10 个问题。对于每个问题，我们分别使用两个 MCP 系统运行三次，共进行了 60 次运行。在此对比中，两个 MCP 系统均使用 Claude Sonnet 4.5，并采用相同的超时设置（1200 秒）。结果如表 2 所示：ipython-mcp 达到了

100% 的准确率（30/30 次运行），而 mcp-solver-asp 的准确率为 90%（27/30 次），在积木世界、资源分配和量子电路问题上未能成功。我们还发现，ipython-mcp 使用的工具调用次数显著更少（平均 16.0 次，而 mcp-solver-asp 为 47.2 次），耗时也更短（216 秒，而 mcp-solver-asp 为 297 秒）。

讨论 这种性能差异可以从架构权衡的角度解释。ipython-mcp 方法能够轻松支持完整的 Python 表达能力，因为智能体可以将 ASP 程序构造成字符串，使用循环编写重复约束，实现自定义解析和解提取，并利用 Python 的调试功能。mcp-solver-asp 方法则强制采用逐项模型构建，这需要调用更多工具，并限制了过程式抽象。不过，mcp-solver-asp 的专用接口也有一些本次比较未体现的潜在优势：基于条目的编辑支持更细粒度的撤销/重做功能，使模型状态可以被明确检查，并能避免某些类型的错误，例如嵌入式 ASP 中的字符串引号问题。它还包含代码分析功能，可防止可能有害的代码运行；而 ipython-mcp 没有固定的防护机制。在教学场景或交互式探索中，声明式接口可能更直观。

8 结论

我们介绍了 ASP-Bench，这是一个用于评估从自然语言到答案集编程的自动翻译的基准测试，包含 128 个问题实例（64 个基础问题，各有简单和困难变体）。该基准测试系统地覆盖了核心 ASP 构造，并支持语义验证，以检查解是否满足问题规范。

基于 ReAct 框架的智能体方法在多次运行中，均使简单和困难问题集达到完全饱和。这凸显了结合求解器反馈进行迭代细化的有效性。这种方法并不局限于固定流程。

尽管已经实现完全饱和，我们仍认为该基准为神经符号人工智能研究提供了宝贵基础。它为研究人员提供了一个严格的测试平台，涵盖多种推理模式、语义验证，以及用于识别系统在特定 ASP 特性上失效原因的诊断能力。具体而言，该基准可用于比较不同方法在多种效率指标上的表现，包括令牌消耗、执行周期和实际耗时。

下一代自然语言到 ASP 系统应在保持正确性和准确性的同时，力求减少推理—行动周期的数量和令牌消耗。此类改进可通过使用 DSPy 等系统进行自动提示优化来实现 [12],；这也有助于开展消融研究，以确定项目提示中的哪些部分对智能体性能贡献最大。单体式项目提示也可采用渐进式披露：智能体根据问题特征选择性地加载相关部分，在保留专门指导的同时减少令牌消耗。

此外，ASP-Bench 可用于评估较小的开放权重模型、提示优化系统以及其他智能体架构，在这些场景中正确性可能无法达到饱和。另一个方向是使用可本地部署的开放权重大语言模型实现相当的覆盖范围。这使得自动化答案集编程无需依赖商业 API 服务即可实现。

困难基准问题连同参考验证器的设计是一项非平凡的任务，这是我们在设计 ASP-bench 的困难问题时认识到的事实。未来工作中一个具有挑战性的目标是设计一类新的“非常困难”问题，并将其纳入更新后的 ASP-Bench。

除了生成正确的答案集编程编码之外，神经符号方法还可以通过探索不同的约束形式化方式并找出更高效的问题表示，针对求解器性能优化编码；我们希望 ASP-Bench 也能在这一方向上提供坚实的基础。

参考文献

[1] Anthropic。《模型上下文协议》，2024年。用于将语言模型连接到外部系统的协议规范。网址：https://modelcontextprotocol.io。

[2] Shaun Lee Baek、Shaun Esua-Mensah、Cyrus Tsui、Sejan Vigneswaralingam、Abdullah Alali、Michael Lu、Vasu Sharma和Kevin Zhu。Rosetta-PL：命题逻辑作为大语言模型推理的基准。见Abteen Ebrahimi、Samar Haider、Emmy Liu、Sammar Haider、Maria Leonor Pacheco和Shira Wein编，*2025*年计算语言学协会美洲国家分会会议论文集：人类语言技术，*NAACL 2025*——第*4*卷：学生研究研讨会，美国新墨西哥州阿尔伯克基，*2025*年*4*月*30*日至*5*月*1*日，第551–562页。计算语言学协会，2025。URL: https://doi.org/10.18653/v1/2025.naacl-srw.53，doi:10.18653/V1/2025.NAACL-SRW.53。

[3] Simone Caruso、Carmine Dodaro、Marco Maratea、Marco Mochi 和 Francesco Riccio。CNL2ASP：将受控自然语言句子转换为答案集编程。理论与实践逻辑编程，24(2)：196–226，2024年。URL：https://doi.org/10.1017/s1471068423000388，doi：10.1017/S1471068423000388。

[4] Erica Coppolillo、Francesco Calimeri、Giuseppe Manco、Simona Perri 和 Francesco Ricca。LLASP：针对答案集编程微调大语言模型。载于 Pierre Mar- quis、Magdalena Ortiz 和 Maurice Pagnucco 编，第 *21* 届知识表示与推理原理国际会议论文集，*KR 2024*，越南河内。*2024*年 *11* 月 *2* 日至 *8* 日，2024年。URL: https://doi.org/10.24963/kr.2024/78，doi：10.24963/KR.2024/78。

[5] William English、Chase Walker、Dominic Simon、Sumit Kumar Jha 和 Rickard Ewetz。可验证的自然语言到线性时序逻辑翻译：一个基准数据集和评估套件。*CoRR*，abs/2507.00877，2025年。网址：https://doi.org/10.48550/arXiv. 2507.00877，arXiv:2507.00877，doi:10.48550/ARXIV.2507.00877。

[6] Martin Gebser、Roland Kaminski、Benjamin Kaufmann 和 Torsten Schaub。实践中的答案集求解。《人工智能与机器学习综合讲义》。Morgan & Claypool Publishers，2012年。doi:10.2200/S00457ED1V01Y201211AIM019。

[7] Martin Gebser、Roland Kaminski、Benjamin Kaufmann 和 Torsten Schaub。使用 clingo 进行多轮 ASP 求解。理论与实践逻辑程序，19(1):27–82，2019年。doi:10.1017/ S1471068418000054。

[8] Michael Gelfond 和 Vladimir Lifschitz。逻辑编程的稳定模型语义。载于 Robert A. Kowalski 和 Kenneth A. Bowen 编，《逻辑编程：第五届国际会议暨研讨会论文集，1988年8月15日至19日，美国华盛顿州西雅图》（共2卷），第1070–1080页。麻省理工学院出版社，1988年。

[9] Simeng Han, Hailey Schoelkopf, Yilun Zhao, Zhenting Qi, Martin Riddell, Wenfei Zhou, Ja mes Coady, David Peng, Yujie Qiao, Luke Benson, Lucy Sun, Alexander Wardle-Solano, Hannah Szabó, Ekaterina Zubova, Matthew Burtell, Jonathan Fan, Yixin Liu, Brian Wong, Malcolm Sailor, Ansong Ni, Linyong Nan, Jungo Kasai, Tao Yu, Rui Zhang, Alexander R. Fabbri, Wojciech Kryscinski, Semih Yavuz, Ye Liu, Xi Victoria Lin, Shafiq Joty, Yingbo Zhou, Caiming Xiong, Rex Ying, Arman Cohan 和 Dragomir Radev。《FOLIO：基于一阶逻辑的自然语言推理》。载于 Yaser Al-Onaizan、Mohit Bansal 和 Yun- Nung Chen 编，*2024*年自然语言处理实证方法会议论文集，*EMNLP 2024*，美国佛罗里达州迈阿密，*2024*年*11*月*12*—*16*日，第22017– 22031页。计算语言学协会，2024年。网址：https://doi.org/10.18653/ v1/2024.emnlp-main.1229, doi:10.18653/V1/2024.EMNLP-MAIN.1229。

[10] Xinyi Hou、Yanjie Zhao、Shenao Wang 和 Haoyu Wang。《模型上下文协议（MCP）：现状、安全威胁与未来研究方向》。*CoRR*，abs/2503.23278，2025年。arXiv:2503.23278，doi:10.48550/arXiv.2503.23278。[11] Adam Ishay、Zhun Yang 和 Joohyung Lee。利用大语言模型生成答案集程序。载于 Pierre Marquis、Tran Cao Son 和 Gabriele Kern-Isberner 编，第*20*届知识表示与推理原理国际会议论文集，*KR 2023*，希腊罗得岛，*2023*年*9*月*2—8*日，第374–383页，2023年。网址：https://doi.org/10.24963/kr.2023/37，doi:10.24963/KR.2023/37。[12] Omar Khattab、Arnav Singhvi、Paridhi Maheshwari、Zhiyuan Zhang、Keshav Santhanam、Sri Vardhamanan、Saiful Haq、Ashutosh Sharma、Thomas T. Joshi、Hanna Moazam、Heather Miller、Matei Zaharia 和 Christopher Potts。《DSPy：将声明式语言模型调用编译为最先进的流水线》。载于第十二届国际学习表征会议，*ICLR 2024*，奥地利维也纳，*2024*年*5*月*7—11*日。OpenReview.net，2024年。网址：https://openreview.net/forum?id=sY5N0zY5Od。[13] Nicola Leone、Gerald Pfeifer、Wolfgang Faber、Thomas Eiter、Georg Gottlob、Simona Perri 和 Francesco Scarcello。《用于知识表示与推理的DLV系统》。*ACM*计算逻辑汇刊，7(3):499–562，2006年。doi:10.1145/1149114.1149117。[14] Kostis Michailidis、Dimos Tsouros 和 Tias Guns。《CP-Bench：评估大语言模型的约束建模能力》。*CoRR*，abs/2506.06052，2025年。网址：https://doi.org/10. 48550/arXiv.2506.06052，arXiv:2506.06052，doi:10.48550/ARXIV.2506.06052。[15]Manuel A. Borroto Santana、Irfan Kareem 和 Francesco Ricca。《从自然语言规范自动组合答案集编程程序》。载于第三十三届人工智能国际联合会议论文集，*IJCAI 2024*，韩国济州岛，*2024*年*8*月*3—9*日，第6198–6206页。ij cai.org，2024年。网址：https: //www.ijcai.org/proceedings/2024/685。[16] Arseny Skryagin、Daniel Ochs、Phillip Deibert、Simon Kohaut、Devendra Singh Dhami 和 Kristian Kersting。《答案集网络：将答案集编程转化为深度学习》。*CoRR*，abs/2412.14814，2024年。网址：https://doi.org/10.48550/arXiv.2412.14814，arXiv:2412.14814，doi:10.48550/ARXIV.2412.14814。[17] Stefan Szeider。《ASP-Bench：问题、真实值与解》，2025年。doi:10.5281/ zenodo.18062939。[18]Stefan Szeider。《通过模型上下文协议连接语言模型与符号求解器》。载于 Jeremias Berg 和 Jakob Nordström 编，第*28*届可满足性测试理论与应用国际会议，*SAT 2025*，*2025*年*8*月*12—15*日，苏格兰格拉斯哥，第341卷，*LIPIcs*，第30:1–30:12页。Schloss Dagstuhl - Leibniz- Zentrum für Informatik，2025年。网址：https://doi.org/10.4230/LIPIcs.SAT.2025.30，doi:10.4230/LIPICS.SAT.2025.30。[19] Stefan Szeider。《CP-Agent：智能体约束编程》。载于代码大语言模型国际研讨会（*LLM4Code*），巴西里约热内卢，2026年4月。arXiv预印本arXiv:2508.07468。[20] Zhun Yang、Adam Ishay 和 Joohyung Lee。《NeurASP：将神经网络融入答案集编程》。载于 Christian Bessiere 编，第二十九届人工智能国际联合会议论文集，*IJCAI 2020*，第1755–1762页。ijcai.org，2020年。网址：https://doi.org/10.24963/ijcai.2020/243，doi:10.24963/IJCAI.2020/243。

[21] Shunyu Yao、Jeffrey Zhao、Dian Yu、Nan Du、Izhak Shafran、Karthik R. Narasimhan 和 Yuan Cao。ReAct：在语言模型中协同推理与行动。载于第十一届国际学习表征会议，*ICLR 2023*，基加利，卢旺达，*2023*年*5*月*1*日至*5*日。OpenReview.net，2023。网址：https://openreview.net/forum?id=WE_vluYUL-X。