软件学报 ISSN 1000-9825, CODEN RUXUEW
2025,36(8):3477−3493 [doi: 10.13328/j.cnki.jos.007346] [CSTR: 32375.14.jos.007346]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

基于记忆策略的元解释学习*

王 榕[1], 田 聪[1], 孙 军[2], 于 斌[1], 段振华[1]

[1](西安电子科技大学 计算机科学与技术学院, 陕西 西安 710126)

[2](School of Computing and Information Systems, Singapore Management University, Singapore 188065, Singapore)

通信作者: 王榕, E-mail: rwang_bily@stu.xidian.edu.cn

摘　要: 元解释学习 (meta-interpretive learning, MIL) 是一种归纳逻辑程序设计 (inductive logic programming, ILP) 方法, 旨在从一组实例、元规则和其他背景知识中学习一个程序. MIL 采用深度优先和失败驱动策略在程序空间中搜索适当的子句以生成程序. 事实上, 这种机制不可避免地引发了对相同目标重复证明的问题. 提出一种剪枝策略, 该策略利用 Prolog 内置的数据库机制来存储未能达成的目标及其对应的错误信息, 从而有效避免冗余的证明过程. 此后, 这些累积的错误信息能够作为指导, 帮助 MIL 系统在未来的学习过程中进行优化和调整. 证明剪枝算法的正确性, 并在理论上计算程序空间的缩减比例. 将所提出的方法应用于两个现有的 MIL 系统 Metagol 和 $\text{Metagol}_{\text{AI}}$, 从而产生了两个新的 MIL 系统 $\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$. 在 4 个不同任务上的实证结果表明, 所提出的策略可以显著减少学习相同程序的时间消耗.

关键词: 元解释学习; 冗余证明; 记忆策略; 剪枝算法; 归纳逻辑程序设计

中图法分类号: TP18

中文引用格式: 王榕, 田聪, 孙军, 于斌, 段振华. 基于记忆策略的元解释学习. 软件学报, 2025, 36(8): 3477–3493. http://www.jos.org.cn/1000-9825/7346.htm

英文引用格式: Wang R, Tian C, Sun J, Yu B, Duan ZH. Meta-interpretive Learning Based on Memory Strategy. Ruan Jian Xue Bao/Journal of Software, 2025, 36(8): 3477–3493 (in Chinese). http://www.jos.org.cn/1000-9825/7346.htm

Meta-interpretive Learning Based on Memory Strategy

WANG Rong[1], TIAN Cong[1], SUN Jun[2], YU Bin[1], DUAN Zhen-Hua[1]

[1](School of Computer Science and Technology, Xidian University, Xi'an 710126, China)

[2](School of Computing and Information Systems, Singapore Management University, Singapore 188065, Singapore)

Abstract: Meta-interpretive learning (MIL) is an approach within inductive logic programming (ILP) that aims to learn a program from a set of examples, metarules, and other background knowledge. MIL uses a depth-first, failure-driven strategy to explore appropriate clauses in the program space to generate programs. This mechanism, however, inevitably leads to the problem of repeated proofs for the same goals. A pruning strategy is proposed, utilizing Prolog's built-in database mechanism to store failed goals and their corresponding error information, effectively preventing redundant proof processes. The accumulated error information serves as a guide to help the MIL system optimize and adjust its learning process in future iterations. The correctness of the pruning algorithm is proved, and the reduction in program space is calculated theoretically. The proposed method is applied to two existing MIL systems, Metagol and $\text{Metagol}_{\text{AI}}$, resulting in two new MIL systems $\text{Metagol}_{\text{F}}$ and $\text{Metagol}_{\text{AI\_F}}$. Empirical results from four different tasks demonstrate that the proposed strategy significantly reduces the time required to learn the same programs.

Key words: meta-interpretive learning (MIL); redundant proof; memory strategy; pruning algorithm; inductive logic programming (ILP)

* 基金项目: 国家自然科学基金 (62192734)
本文由"形式化方法与应用"专题特约编辑陈明帅研究员、田聪教授、熊英飞副教授推荐.
收稿时间: 2024-08-25; 修改时间: 2024-10-14; 采用时间: 2024-11-26; jos 在线出版时间: 2024-12-10
CNKI 网络首发时间: 2025-04-21

作为符号人工智能 (AI) 的一个分支, 归纳逻辑程序设计 (inductive logic programming, ILP)[1]专注于从实例和背景知识中归纳构建逻辑程序 (即一阶子句理论). ILP 系统使用语言偏误来定义程序空间. 因此, 学习程序的过程可以视为在程序空间内的搜索. Muggleton 等人[2]在 2014 年引入了元解释学习 (meta-interpretive learning, MIL), 这是一种新的 ILP 方法, 相较于早期的 ILP 系统如 FOIL[3]、Progol[4]、TILDE[5]和 Aleph[6]标志着显著的进步. 其主要区分在于支持递归概念和谓词发明的学习, 使其与前身系统有所不同.

然而, MIL 通过引入元规则来限制可学习程序的形式, 从而更精确地定义了程序空间. 在 MIL 学习者中, 该方法采用迭代加深深度优先搜索 (iterative deepening depth-first search, IDDFS) 来导航这个定义好的程序空间. 该方法虽然创新, 但在搜索庞大的程序空间时带来了挑战, 使得 MIL 变慢且容易生成冗余证明, 这是其固有学习机制导致的结果. IDDFS 是一种搜索方法, 它不断执行深度限制的深度优先搜索, 逐步增加深度限制, 直到找到目标. 具体来说, MIL 系统通过逐渐增加子句数量来搜索程序空间. 因此, MIL 程序的搜索空间随着程序子句数量的增加呈指数增长. 这种指数扩展不仅导致巨大的搜索空间, 还由于涉及的计算工作量巨大和大量冗余计算, 严重阻碍了 MIL 的实际应用.

为了减轻元解释学习中广泛搜索空间的挑战, 研究人员采用了几种策略来提高 MIL 的搜索效率. 这些改进包括分层偏误重构[7]、通过减少候选解决方案的元规则集来最小化[8,9]、引入多态和精炼类型检查以显著提高元解释学习的效率[10]、修订实例空间和假设空间以提高效率[11]以及与神经网络结合[12,13]. 然而, 这些方法并未从根本上改变 MIL 的学习机制. 因此, 由迭代深度优先搜索产生的冗余证据问题仍未得到解决. 随着程序子句数量的增加, 这个问题变得更加严重, 并可能导致系统崩溃, 因为计算需求呈指数增长. 通过引入基于记忆策略的剪枝算法, 我们旨在解决 MIL 搜索效率的挑战. 该算法通过减少冗余证明显著提高了 MIL 系统的计算效率, 从而最小化计算资源的浪费并节省能源. 这一改进使系统能够更有效地处理复杂的学习任务.

MIL 采用由失败驱动的深度优先遍历搜索. 这种方法将 SLD-树中从给定 Goal 开始的所有路径标记为失败分支, 直到发现一个可行的程序. 基于此基础, 我们在 MIL 中集成了基于记忆策略的剪枝算法. 该算法旨在捕捉和利用在学习过程中遇到的目标的错误信息. 借此, 我们在 Prolog 中实现了一个动态数据库, 用于归档所有目标及其相关的错误信息. 一旦某个目标被认为失败, 那么在尝试证明新的 Goal 之前, 系统会首先评估该 Goal 是否已被记录在数据库中. 如果发现匹配, 则 MIL 学习者放弃重新评估已知的错误目标, 而是立即进行回溯. 如果该目标是新的或以前未遇到的, 则将其连同其错误数据一起添加到数据库中以供将来参考. 这确保了在学习过程的范围内, 每个 Goal 最多只进行一次证明, 从而显著简化了系统的效率.

本文的主要贡献如下.

我们通过一个基于内存的剪枝算法, 标记出一个直接解决方案来提高 MIL 的效率, 从而识别并解决了 MIL 的冗余问题, 这是由其 IDDFS 策略引起的. 我们介绍了剪枝算法并证明了其正确性和有效性, 展示了程序空间在理论上的减少. 通过将该算法应用于现有的 MIL 系统, 我们展示了其实际价值, 并获得了两个更新的 MIL 系统 ($\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$), 我们的实验证实了它在减少学习时间方面的有效性.

本文第 1 节介绍元解释学习的预备知识和框架. 第 2 节使用一个激励性的例子来说明 MIL 中的冗余证明问题. 第 3 节介绍针对 MIL 的修剪算法并证明该方法的正确性. 第 4 节介绍两个改进的 MIL 系统——$\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$. 第 5 节给出了 4 个学习任务的实验结果. 第 6 节回顾相关工作. 最后, 第 7 节总结本文并概述未来的工作方向.

1 基础知识

本节首先介绍本文使用的逻辑符号以及 MIL 的理论框架.

1.1 逻辑学符号

我们考虑一个具有固定词汇的谓词符号 Π、函数符号 $\mathcal{F}$ 和变量 $\mathcal{V}$ 的一阶语言. $\mathcal{T}(\mathcal{F},\mathcal{V})$ 表示使用来自 $\mathcal{F}$ 的符号和来自 $\mathcal{V}$ 的变量构造的项的集合. 通过在一个 n 元谓词 $p/n \in \Pi$ 上应用替换 $\theta = \{v_1 \mapsto t_1,\ldots,v_n \mapsto t_n\}$, 可以得到形式为 $p(t_1,\ldots,t_n)$ 的原子, 其中 $t_i \in \mathcal{T}(\mathcal{F},\mathcal{V})$, 并且 θ 是对 $i = 1,\ldots,n$ 的变量 v_i 的项 t_i 的赋值. 所有谓词符号的集合称为谓词签名. 对于一个原子 A, A 及其否定 $\neg A$ 都是文字. 一条子句由有限集合 (可能为空) 的文字表示, 记作

$\{A_1,A_2,\ldots,\neg A_i,\neg A_{i+1},\ldots\}$ 或 $A_1,A_2,\ldots \leftarrow A_i,A_{i+1},\ldots$. 这里, 正文字构成正体. 特别地, 一个具有空正体的子句称为纯负子句. 一个霍恩子句 (*Horn* 子句) 是一个最多包含一个正文字的子句. 一个确定子句形式为 $head \leftarrow body$ (在 Prolog 中记作 $Head:-Body$), 其中包含恰好一个正文字. 对于给定的确定子句, 其正文字 $head$ 是一个原子, 负文字的集合 $body \equiv (a_1 \wedge \ldots \wedge a_n)$ 是一个目标 (原子的合取). 注意, 空目标表示为 true. 一个确定程序是一组有限的确定子句. 以下, 我们聚焦于确定程序. 如果一个程序包含至少一个作为另一个谓词参数的谓词符号, 那么它被称为高阶程序. 其他未提及的术语可参见文献 [14,15].

MIL 基于 SLD-解析 (SLD-resolution)[16]学习逻辑程序. 在 SLD-解析方法中, 通过找到一种 SLD-驳斥[17]来证明一组 Horn 子句不可满足.

定义 1 (SLD-推导). 设 S 是由一组确定子句 D 和一组目标 $\{G_1,\ldots,G_q\}$ 组成的 Horn 子句集. 一个 SLD-推导对于 S 是一系列的否定子句 $\langle N_0,N_1,\ldots,N_p\rangle$, 满足以下所有性质:

(1) $N_0 = G_j$, 其中 G_j 是目标之一.

(2) 对于序列中的每一个 $N_i (0 \leqslant i < p)$, 如果 $N_i = \leftarrow A_1,\ldots,A_{k-1},A_k,A_{k+1},\ldots,A_n$, 那么在 D 中存在一个确定子句 $C_i = A_k \leftarrow B_1,\ldots,B_m$, 使得当 $m>0$ 时, $N_{i+1} = \leftarrow A_1,\ldots,A_{k-1},B_1,\ldots,B_m,A_{k+1},\ldots,A_n$; 并且当 $m=0$ 时, $N_{i+1} = \leftarrow A_1,\ldots,A_{k-1},A_{k+1},\ldots,A_n$.

定义 2 (SLD-驳斥). 当且仅当 $N_p = \square$ (空子句) 时, SLD 推导是一个 SLD-驳斥.

定义 3 (SLD-解式). 设 $G_0 = \leftarrow A_0,\ldots,A_m$ 为一个目标, $C = B_0 \leftarrow B_1,\ldots,B_n$ 为一个 Horn 子句, 其中文字 A_i 在替换 θ 下与 B_0 是可统一的. 目标 G_1 (称为 SLD-解式) $G_1 = \leftarrow A_0,\ldots,A_{i-1},B_1,\ldots,B_n,A_{i+1},\ldots,A_m$ 被认为是在替换 θ 下, 通过 C 从 G_0 推导出来的.

SLD-解析可以使用任何选择规则来选择目标中的文字. Prolog 总是选择目标中的最左边的文字[18]. 相对于一个确定的程序 P, 目标 G_0 可能有多个 SLD-推导, 因为 G_0 可以与 P 中的多个 Horn 子句进行解析. 我们将所有可能的推导表示为一个 SLD-树.

定义 4 (SLD-树). 设 P 为一个确定性程序, G_0 为初始目标. $P \cup \{G_0\}$ 的 SLD-树是一个 (可能是无限的) 树, 其中每个节点是一个目标, 两个节点之间的边表示它们之间的 SLD-推导.

定义 5 (成功分支[19]). 设 P 是一个确定程序, G_0 是一个初始目标, T 是 $P \cup \{G_0\}$ 的 SLD-树. 成功分支是根 (G_0) 与包含空子句的叶子之间的路径.

定义 6 (失败分支). 设 P 为一个确定程序, G_0 为一个初始目标, T 为 $P \cup \{G_0\}$ 的一个 SLD-树. 失败分支是指从根 (G_0) 到一个包含非空目标的叶节点的一条路径.

1.2 元解释学习

通过利用经过改编的 Prolog 元解释器, MIL 方法通过证明一组目标来学习逻辑程序. 一个标准的 Prolog 元解释器通过反复获取其头部与目标匹配的一阶子句来证明目标. 在此基础上, MIL 学习者通过额外获取其头部与目标匹配的高阶元规则来证明目标. 一个 MIL 问题由 $Input = <B,E,MS>$ 和 $Output = H$ 组成. B 表示用户提供的背景知识. $E = <E^+,E^->$ 表示一对实例集, 其中 E^+ 和 E^- 分别表示正例集和负例集. MS 表示元规则集. 这里, 元规则被表示为如表 1 所示的高阶 Horn 子句.

表 1 元规则示例

名称	元规则	阶约束
Curry3	$P(x,y) \leftarrow Q(x,y,R)$	$P \succ Q,\ P \succ R$
Curry4	$P(x,y) \leftarrow Q(x,y,R,S)$	$P \succ Q,\ P \succ R,\ P \succ S$
Precon	$P(x,y) \leftarrow Q(x),R(x,y)$	$P \succ Q,\ P \succ R$
Postcon	$P(x,y) \leftarrow Q(x,y),R(y)$	$P \succ Q,\ P \succ R$
Chain	$P(x,y) \leftarrow Q(x,z),R(z,y)$	$P \succ Q,\ P \succ R$
Tailrec	$P(x,y) \leftarrow Q(x,z),P(z,y)$	$P \succ Q,\ x \succ z \succ y$

例如, 在 *chain* 元规则中 $P(x,y) \leftarrow Q(x,z),R(z,y)$, 大写字母 P,Q 和 R 表示存在量化的二阶变量, 而小写字母 x, y 和 z 表示全称量化的一阶变量. 每个元规则都与一个阶 (order) 约束相关联, 以确保证明过程的终止. H 表示通过将元替换 $\Sigma=\{v_1/t_1,\ldots,v_n/t_n\}$ 应用到其对应元规则而形成的假设, 其中, 存在量化变量 v_i 被替换为项 t_i. MIL 的任务是学习一个假设 H 使得对于 $\forall e^+\in E^+,B,H\models E^+$ 并且对于 $\forall e^-\in E^-,B,H\not\models e^-$. 这里 $\models$ 表示蕴涵.

2 问题: 冗余证明

在本节, 我们使用一个具有激励意义的例子“青蛙与荷叶问题”来揭示 MIL 中的冗余证明问题. 青蛙与荷叶问题的任务描述如下: 有 $2n+1$ 片荷叶和 $2n$ 只青蛙, 其中包括 n 只黄色青蛙和 n 只绿色青蛙. 我们假定每片荷叶最多只容纳一只青蛙. n 只绿色青蛙蹲在最左边的 n 片荷叶上, 其余 n 只黄色青蛙蹲在最右边的 n 片荷叶上, 中间的荷叶是空的. 由于荷叶之间的距离限制, 每只青蛙可采用以下两种方式跳跃:

1) 如果青蛙与空荷叶相邻, 它可以直接跳到空荷叶上;

2) 如果青蛙不与空荷叶相邻, 它只能跨过一只青蛙跳到空荷叶上.

目标是找到一种跳跃策略, 以最少的跳跃步数完成在黄蛙和绿蛙之间的位置交换. 例如, 当 $n=2$ 时, 任务的初始状态和最终状态如图 1 所示.

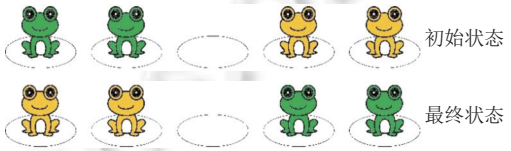


图 1 青蛙与荷叶问题的初始状态和最终状态, 其中 $n=2$

我们将绿色青蛙、黄色青蛙和空荷叶分别表示为 1、−1 和 0. 因此, 这个问题的初始状态和最终状态可以表示为列表 $[1,1,0,-1,-1]$ 和 $[-1,-1,0,1,1]$. 注意, 青蛙的移动等同于空荷叶的移动. 针对这个任务, 我们设计了 4 个基本的 2-元谓词 right_one/2、right_two/2、left_one/2 和 left_two/2 (简称为 $r1/2$、$r2/2$、$l1/2$ 和 $l2/2$) 来描述空荷叶的移动. 这些谓词表示空荷叶向其左或右移动了 1 或 2 个位置. 数字 2 表示这些谓词有两个参数 (输入和输出). 例如在图 2 中, 谓词 right_one/2 表示空荷叶移动到了其右侧的第 1 个位置, 这也意味着空荷叶右侧的第 1 只青蛙跳到了空荷叶上. 因此, 该任务可以描述为一个原子 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$. 我们需要基于可用的背景知识学习原子 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 的定义. 此实验使用链式元规则 $P(X,Y):-Q(X,Z),R(Z,Y)$. 青蛙与荷叶问题的具体输入设置如表 2 所示.

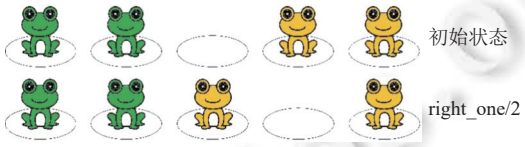


图 2 由 2-元谓词 right_one/2 引起的状态变化

表 2 青蛙和荷叶问题的输入设置.

输入	设置
实例集	$[f([1,1,0,-1,-1],[-1,-1,0,1,1])]$
元规则集	$[P(X,Y):-Q(X,Z),R(Z,Y)]$
背景知识	$r1/2$, $r2/2$, $l1/2$, $l2/2$

通用元解释器的理论框架如代码 1 所示. 为了生成一个程序, MIL 系统首先尝试通过 ($call(Atom)$) 将 *Atom* 作为目标进行证明. 当证明失败时, MIL 系统尝试将目标与元规则的头部匹配, 检查 Order 约束, 并将元规则中存在量化的变量绑定到谓词签名中的符号. MIL 系统保存产生的元替换 *MetaSub* 并尝试证明元规则的体目标. 在证明

所有目标后, 通过将保存的元替换投影到其对应的元规则上来构建假设.

代码 1. 通用元解释器的 Prolog 代码.

```
prove([], G, G).
prove(Atom|Atoms,G1,G2) :-
    prove_aux(Atom,G1,G3),
    prove(Atoms,G3,G2).
prove_aux(Atom,G,G) :-
    call(Atom).
prove_aux(Atom,G1,G4) :-
    metarule(Name,MetaSub,(Atom :- Body)),
    OrderTest,
    save_subset(metasub(Name,MetaSub),G1,G3),
    prove(Body,G3,G2).
```

对于“青蛙与荷叶问题”, MIL 首先尝试用 1 条子句和 0 个发明谓词来证明原子 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$. 当证明失败时, MIL 尝试通过增加子句 N 和发明谓词 M 的数量来证明该原子 $(0 \leqslant M \leqslant N-1)$. 具体来说, 根据元规则的格式, 初始目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 可以分解为两个体原子. 然后这两个体原子将以类似的方式作为新目标进行证明. 一旦路径被证明为失败分支, MIL 学习者将回溯以证明下一个目标, 直到找到成功分支为止.

图 3 显示了目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 在 $N=2$, $M=1$ 情况下 SLD-树示意图. SLD-树展示了所有可能的包含 2 条子句和 1 个发明的 2 元谓词 ($f1/2$) 的程序. 图 3 中的蓝色子树表示当第 1 条子句已知为 $f([1,1,0,-1,-1],[-1,-1,0,1,1]):-r1([1,1,0,-1,-1],[1,1,-1,0,-1]),f1([1,1,-1,0,-1],[-1,-1,0,1,1])$ 时程序第 2 条子句的解空间, 如果 MIL 在完成包含两条子句的程序空间搜索后未能找到解, 则它将逐步增加子句的数量和发明谓词的数量. 如图 4 所示的 SLD-树示意图, MIL 将重新从初始目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 开始搜索, 使用 3 条子句和 2 个发明的谓词. 从这个例子可以看到, 图 4 中的灰色子树与图 3 中的蓝色子树完全相同. 这意味着, 目标 $f1([1,1,-1,0,-1],[-1,-1,0,1,1])$ 将会被重复证明.

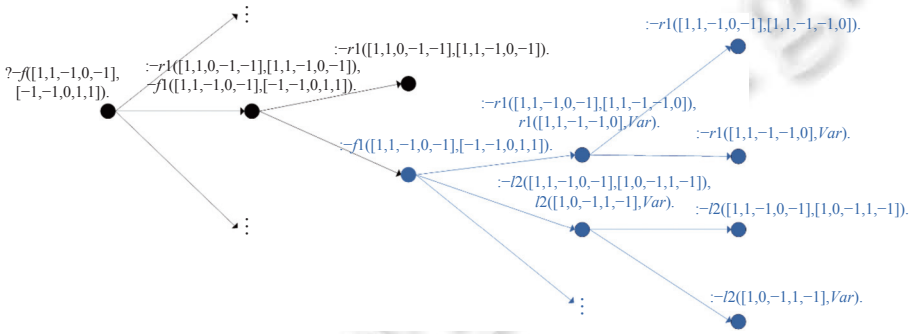


图 3 当 $N=2, M=1$ 时, 目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 的 SLD-树的示意图

需要指出的是, 还有另一种包含未知变量的目标类型 (例如, 图 4 中的目标 $f2([1,1,0,-1,-1],Var)$, 其中变量 Var 在 Prolog 中被称为匿名变量). 这些目标也将在 MIL 的后续学习过程中被重复证明. 例如, 图 5 中的灰色子树与图 4 中的红色子树完全相同.

因此, 可以合理地认为冗余证明是 MIL 中存在的问题. 基于此动机, 我们提出了一种剪枝策略, 以避免重复证明相同的目标.

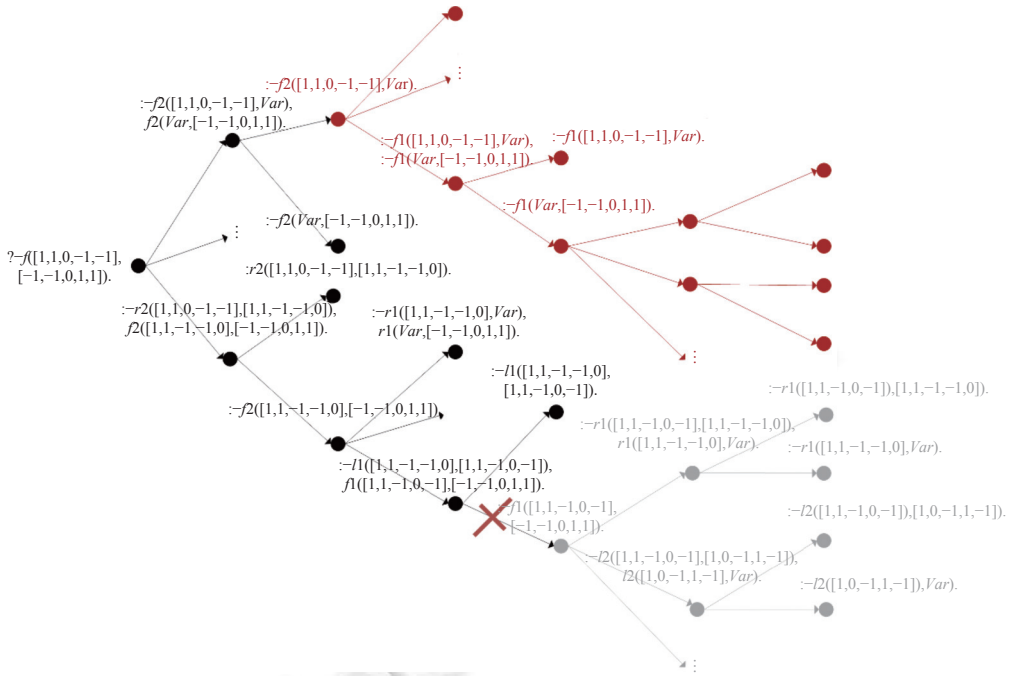


图 4 当 $N=3, M=2$ 时, 目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 的 SLD-树的示意图

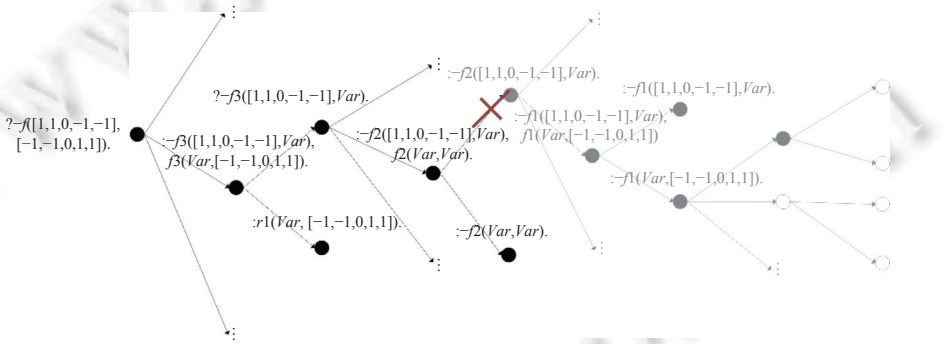


图 5 当 $N=4, M=3$ 时, 目标 $f([1,1,0,-1,-1],[-1,-1,0,1,1])$ 的 SLD-树示意图

3 元解释学习剪枝算法

如算法 1 所示, 为了在元解释学习 (meta-interpretive learning, MIL) 中实现剪枝, 我们的方法利用了保存失败目标能划定搜索空间中不可行区域的洞察. 通过记录这些失败, 算法避免了冗余探索, 而是专注于有前景的区域, 从而提高搜索效率. 这个原理是我们基于记忆的剪枝算法开发的基础, 通过将 SLD-树中的节点分类为常量化目标和非常量化目标, 为更高效的 MIL 过程奠定了基础.

算法 1. MIL 剪枝算法.

输入: 实例集 E, 背景知识 BK, 元规则集 MS, 数据库 DB, 谓词集 Sig, 最大子句数 N, 最大发明谓词数 $M=N-1$;
输出: 程序 H.

1. 初始化: $DB \leftarrow \varnothing, H \leftarrow \varnothing$
2. for $Atom \in E$ do
3. for $n \in [1, N]$ do
4. for $m \in [0, M]$ do
5. get Sig
6. $Depth = n - length(H)$
7. while $Atom = [P|Args] \in SLD\text{-}tree$ do
8. if $Atom$ 是一个常量化目标
9. $Goal_f = [Depth, Sig, P|Args]$
10. else
11. $extract(Atom, [P|Args'])$
12. $Goal_f = [Depth, Sig, P, Args'|H]$
13. end if
14. 检查 $faulty(Goal_f)$ 是否已经在失败数据库 DB 中
15. if $faulty(Goal_f) \in DB$
16. 失败并回溯
17. else
18. $DB \leftarrow DB \cup \{faulty(Goal_f)\}$
19. 证明 $Atom$
20. if true
21. $H \leftarrow H \cup clause(Atom : -Body)$
22. 证明子句体中的下一个目标
23. else
24. 失败并回溯
25. end if
26. end if
27. end while
28. end for
29. end for
30. end for
31. return H

定义 7 (常量化目标和非常量化目标). 设 P 为一个确定性程序, G_0 为初始目标, T 为 $P \cup \{G_0\}$ 的 SLD-树. T 上的每一个原子目标都需要被证明. 当原子目标不包含任何变量时, 称其为常量化目标. 当其包含匿名变量时, 称其为非常量化目标.

为了更好地理解常量化目标和非常量化目标之间的区别, 考虑图 3 和图 4 中描述的两个例子. 在图 3 中, 蓝色子树的根节点被表示为 $f1([1,1,-1,0,-1],[-1,-1,0,1,1])$, 作为一个常量化目标的例子. 这是因为它完全由具体值组成, 没有任何变量, 表明目标中的所有参数都是确定的. 相反, 在图 4 中, 红色子树的根节点被表示为 $f2([1,1,0,-1,-1], Var)$, 代表了一个非常量化目标. 这里, 变量 Var 的存在意味着目标中的并非所有参数都是指定的, 这使得它包含了未知元素.

对于给定的目标, SLD-树的路径将按深度优先顺序进行探索, 直到返回一个成功的分支. 受此启发, 在迭代过程中提取失败分支的失败信息以避免冗余证明是合理的. 相应地, 我们在 MIL 中设计了一种记忆策略, 从 SLD-树

的失败分支中提取失败信息.

本文提出的剪枝算法在算法 1 中描述. 我们首先确定学习的程序的最大子句数 N 和发明谓词 M. 我们通过搜索其对应的 SLD-树来证明每个例子 $Atom \in E$. 在探索过程中, 如果要被证明的 $Atom = [P|Args]$ 是一个常量化目标, 我们提取其失败信息为 $Goal_f = [Depth, Sig, P|Args]$, 其中 $Depth$ 表示剩余子句的数量, Sig 代表谓词签名. 否则, 如果 $Atom$ 是一个非常量化目标, 我们首先通过子句 $extract([P|Args], [P|Args'])$ 提取 $Atom$, 该子句用统一符号 v 替换 $Args$ 中的所有匿名变量以得到 $Args'$. 然后, 我们提取其失败信息为 $Goal_f = [Depth, Sig, P, Args'|H]$. 注意, 在一阶系统中有一个附加的剪枝条件以防止正确的程序被移除. 在第 4.1 节中将解释在一阶系统中引入此条件的原因.

随后, 算法 1 首先检查动态 $faulty$ 数据库 DB 中是否已经记录了子句 $faulty(Goal_f)$. 一旦在数据库中发现该子句, MIL 学习器将跳过证明过程并回溯到前一步以证明其他目标. 如果数据库中未记录 $Atom$ 的失败信息, 即 $faulty(Goal_f)$, 则应立即将其添加到数据库中, 并在下一步中证明 $Atom$. $Atom$ 的证明过程由通用元解释器完成. 一旦所有目标都被证明, 一个一致的假设 H 将被完成并返回. 在生成假设后, 失败数据库中的所有事实或子句将被移除.

定理 1 (算法 1 的正确性). 给定背景知识 B、实例集 $E =< E^+, E^- >$ 和元规则集 MS, MIL 的任务是找到一个程序 H 使得 $B, H \models E^+$ 并且对于 E^- 中的所有 $e^-, B, H/ \models e^-$. 可以断言, 算法 1 的操作不会从程序空间中删除正确的程序 H.

证明: 我们通过数学归纳法证明该定理. 设 N 为最大子句数, $M = N - 1$ 为最大发明谓词数, S 为子句理论集, 即程序空间. 根据 MIL 的深度优先搜索策略, 程序空间 S 可以分为 $S_{1,0} \cup S_{2,0} \cup S_{2,1} \cup \ldots \cup S_{N,0} \cup \ldots \cup S_{N,N-1}$. 其中元素 $s \in S_{i,j} (1 \leqslant i \leqslant N, 0 \leqslant j \leqslant M)$ 是具有 i 条子句和 j 个发明谓词的程序.

情况 1. 首先考虑 $N = 1$ 的情况. 在这种情况下, 程序空间为 $S = S_{1,0}$. 算法 1 在这种情况下不起作用. 因此程序空间不会减少, 正确的程序 H 不会被删除. 因此, 该定理成立.

情况 2. 假设当 $N = k (k \geqslant 2)$ 时, 定理 1 成立. 设程序空间为 $S = S_{1,0} \cup S_{2,0} \cup S_{2,1} \cup \ldots \cup S_{k,0} \cup \ldots \cup S_{k,k-1}$, 所有由算法 1 找到的常量化目标和非常量化目标构成一个失败集 F_k. 因此, 每个元素 $f \in F_k$ 都被证明是失败的. 也就是说, 由失败目标 $f \in F_k$ 生成的程序不可能是正确的程序.

情况 3. 我们考虑 $N = k + 1$ 的情况. 程序空间为 $S = S_{1,0} \cup S_{2,0} \cup S_{2,1} \cup \ldots \cup S_{k,0} \cup \ldots \cup S_{k,k-1} \cup S_{k+1,0} \cup \ldots \cup S_{k+1,k}$. 失败集为 $F_{k+1} = F_k \cup F'$. 根据情况 2, $S_{1,0} \cup S_{2,0} \cup S_{2,1} \cup \ldots \cup S_{k,0} \cup \ldots \cup S_{k,k-1}$ 中由 F_k 的失败目标生成的程序不可能是正确的程序. 现在我们只需要证明由失败目标生成的剩余集 $S_{k+1,0} \cup \ldots \cup S_{k+1,k}$ 中的程序不可能是正确的程序. 设 $s = \{s_1, s_2, \ldots, s_i, \ldots, s_{k+1}\} \in S_{k+1,0} \cup \ldots \cup S_{k+1,k}$ 为 MIL 学到的程序, 其中 s 包含 $k+1$ 条子句且 $s_i = Atom^i : -Body^i$ 是程序的第 i 条子句. 很容易发现, $Atom^i$ 的失败信息不会出现在失败数据库中. 否则, 算法 1 将其识别为失败并进行回溯, $Atom^i$ 无法生成正确的程序.

综上所述, 算法 1 的操作不会删除程序空间中的正确程序.

引理 1 (MIL 假设空间 [20]**)** 给定 p 个谓词符号和 m 个在 M_j^i 中的元规则, 当元规则的体中最多有 j 个文字且每个文字最多有 i 个参数时, 该元规则属于片段 M_j^i. 用 n 条子句表达的程序数目最多为 $(mp^{j+1})^n$.

由于子句数从 1 增加到 n, 我们可以通过引理 1 汇总 MIL 任务搜索空间中的程序数目. 结果 (即 MIL 搜索空间) 被形式化为推论 1.

推论 1 (MIL 搜索空间). 给定 p 个谓词符号和 m 个在 M_j^i 中的元规则, 当元规则的体中最多有 j 个文字且每个文字的参数最多为 i 时, 该元规则属于片段 M_j^i. 由最多 n 条子句组成的所有可能程序的数量为 $\sum_{k=1}^{n} \left(mp^{j+1}\right)^k$.

推论 1 表明 MIL 的搜索空间可以根据程序子句的数量分为多个子空间. 这些子空间中的程序可能包含相同的子程序. 因此, 这些子程序被反复证明和计算.

定理 2 计算剪枝后的 MIL 搜索空间中的程序数目 (其中所有子程序仅被证明一次) 并估计通过剪枝减少的 MIL 搜索空间的比例.

定理 2 (剪枝后的 MIL 搜索空间). 假设有无限量的内存可用, 给定 p 个谓词符号和 m 个在 M_j^i 中的元规则, 一个程序包含最多 n 条子句, 剪枝子树的深度为 l, $l = 1, 2, \ldots, n-1$. 设 $t = mp^{j+1}$, 通过修剪常量化目标和非常量化目

标减少的 MIL 搜索空间比例为 $\frac{\sum_{l=1}^{n}\sum_{k=1}^{n-l}t^k}{\sum_{k=1}^{n}t^k}=\frac{1}{t-1}-\frac{n}{t^n-1}$ 和 $\frac{\sum_{k=1}^{n}\sum_{l=1}^{k-1}t^l}{\sum_{k=1}^{n}t^k}=\frac{1}{t-1}-\frac{n}{t^n-1}$. 当 $t>n$ 时, 方程的极限接近 $\frac{1}{mp^{j+1}-1}$.

证明: 根据推论 1, 在修剪掉所有常量化目标的情况下, MIL 搜索空间中的程序数目为 $\sum_{l=1}^{n}\sum_{k=1}^{n-l}t^k$, 在修剪掉所有非常量化目标的情况下, MIL 搜索空间中的程序数目为 $\sum_{k=1}^{n}\sum_{l=1}^{k-1}t^l$, 而不修剪情况下的 MIL 搜索空间中的程序数目为 $\sum_{k=1}^{n}t^k$. 因此, 通过修剪常量化目标和非常量化目标减少的 MIL 搜索空间比例分别为 $\frac{\sum_{l=1}^{n}\sum_{k=1}^{n-l}t^k}{\sum_{k=1}^{n}t^k}=\frac{1}{t-1}-\frac{n}{t^n-1}$ 和 $\frac{\sum_{k=1}^{n}\sum_{l=1}^{k-1}t^l}{\sum_{k=1}^{n}t^k}=\frac{1}{t-1}-\frac{n}{t^n-1}$. 因此, 当 $t>n$ 时, $\frac{1}{t-1}-\frac{n}{t^n-1}$ 极限值接近于 $\frac{1}{t-1}(t=mp^{j+1})$.

根据定理 2, 假设内存无限可用, 在程序空间非常大且 $mp^{j+1}>n$ 的情况下, 提出的剪枝算法理论上可以将程序空间相对于其原始大小减少到 $\frac{1}{t-1}$(其中 $t=mp^{j+1}$). 这个理论框架说明了剪枝算法通过最小化冗余计算来提高 MIL 效率的潜力. 然而, 这种无限内存的假设与实际计算环境有所偏离, 尤其是在 MIL 系统中实施该策略时.

在 MIL 系统中实际应用此剪枝策略面临主要挑战. Prolog 是一种使用搜索和回溯的编程语言, 通常使用固定的堆栈限制来防止内存溢出, 这与理论模型中假设的内存无限不同. 这个堆栈限制通常设置为默认值, 但也可以自定义, 是一个重要的操作约束. 因此, 由于操作复杂性和假设与可用内存资源之间的固有不匹配, 剪枝算法的实际应用可能无法实现理论上的计算时间减少, 导致理论期望与实际性能之间的差异.

另一个导致学习时间实际减少与程序空间理论减少之间差异的重要原因在于剪枝算法本身所需的计算工作. 算法需要大量计算, 包括检查和更新失败目标. 虽然这些步骤对于识别和消除搜索空间中的冗余路径至关重要, 从而在理论上提高了学习效率, 但它们也增加了 MIL 系统所需的总计算时间. 修剪相关计算所花费的额外时间可能部分抵消预期的效率提升, 从而导致预期理论效益与实际观察结果之间的差异.

4 算法实现

在本节中, 我们将本文法应用于两个 MIL 系统 Metagol 和 $\text{Metagol}_{\text{AI}}$[21], 从而得到两个新的 MIL 系统 $\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$. 需要注意的是, Metagol 是一个一阶 MIL 系统, 而 $\text{Metagol}_{\text{AI}}$[22]是一个通过引入解释背景知识来支持抽象 (abstraction) 和发明 (invention) 的高阶 MIL 系统.

4.1 $\text{Metagol}_{\text{F}}$

$\text{Metagol}_{\text{F}}$ 结合 Metagol 与提出的剪枝算法. 它首先尝试通过 (*call*(*Atom*)) 将一个 *Atom* 证明为目标. 一旦失败, $\text{Metagol}_{\text{F}}$ 尝试通过原始谓词证明 *Atom*. 如果再次失败, $\text{Metagol}_{\text{F}}$ 尝试将目标与一个元规则的头部匹配, 保存元替换, 然后证明主体目标. 在这种情况下, $\text{Metagol}_{\text{F}}$ 将首先尝试使用现有的假设来证明目标. 如果失败, $\text{Metagol}_{\text{F}}$ 将尝试创建一个新的假设 (即新子句). 请注意, 由于一阶程序的表达能力有限, 该程序可能需要多个具有相同头部的子句. Prolog 允许对同一谓词进行多重定义, 表示在不同场景下的不同实现. 虽然这增强了编程语言的灵活性, 但可能导致我们的剪枝算法出现冲突, 因为这些子句具有相同的头部但潜在的不同子节点. 为解决这个问题, 我们在一阶系统 $\text{Metagol}_{\text{F}}$ 中添加了一个额外的修剪条件 (即失败数据库查询). 为了使用新的假设来证明一个 *Atom*, 我们首先从 *Atom* 中提取失败信息, 然后检查失败信息是否已经存在于失败数据库中. 如果是这样, *Atom* 将被立即识别为失败, MIL 学习器将尝试沿另一分支回溯. 如果不是, 我们将 *Atom* 的失败信息添加到失败数据库中, 然后尝试证明 *Atom*.

4.2 $\text{Metagol}_{\text{AI\_F}}$

$\text{Metagol}_{\text{AI\_F}}$ 引入了解释的背景知识以支持抽象和发明. 例如, 代码 2 中所示的高阶谓词 *map*/3 有助于学习比一阶谓词更紧凑的程序. 与 $\text{Metagol}_{\text{AI\_F}}$ 相比, 扩展的 $\text{Metagol}_{\text{F}}$ 系统还尝试将目标与解释的背景知识中的子句头进行匹配.

代码 2. 解释的背景知识 *map*/3.

```
background(([map, [], [], F]:−[]]).
background(([map, [A|As], [B|Bs], F]:−[[F, A, B], [map, As, Bs, F]])).
```

5 实 验

为了评估提出的 MIL 框架剪枝算法的性能, 我们比较了使用和不使用剪枝算法的系统. 实验在 4 个不同的任务上进行: 青蛙与荷叶问题、机器人服务员、国际象棋策略和删除列表尾部. 所有实验都在 Intel Core i9-9900K CPU 和 64 GB 内存的设备上进行.

在评估我们提出的剪枝算法时, 我们的分析重点是比较其与原方法的内存消耗. 为了获得全面且准确的评估, 我们使用了 Prolog 内置谓词 statistics 提供的几个关键性能指标. 这些指标列在表 3 中, 包括推理 (inferences), 即系统的总逻辑推理尝试次数, 原子 (atoms) 表示符号数据的内存消耗, 全局堆栈 (global_stack) 反映了在执行期间的数据结构和调用帧的内存消耗. 我们还监控了 CGC (子句垃圾收集) 来评估程序生命周期内的内存回收效率. 需要注意的是, 在所有进行的实验中, global_stack 都保持在 256 MB, 这符合 SWI-Prolog 的默认堆栈限制.

表 3 评估 Prolog 程序内存消耗的关键指标

指标	定义
Inferences	自Prolog启动以来, 通过调用和重做端口的总次数. 包括子线程中的推理次数
Atoms	内存使用的原子数目
Global_stack	全局使用量, 全局空闲量
CGC	执行的子句垃圾回收次数

5.1 青蛙与荷叶问题

表 4 显示了在有 2 只绿色青蛙和 2 只黄色青蛙时, Metagol 为青蛙与荷叶问题学习到的程序. 生成的程序 (跳跃策略) 包括 5 条子句和 3 个发明谓词 (即 $f3$, $f2$, $f1$).

表 4 青蛙与荷叶问题中 Metagol 学到的程序, 其中 n=2

序号	程序步骤
1	$f(A,B):-f3(A,C),f3(C,B)$
2	$f3(A,B):-f2(A,C),f2(C,B)$
3	$f2(A,B):-f1(A,C),right\_two(C,B)$
4	$f1(A,B):-left\_two(A,C),righ\_one(C,B)$
5	$f2(A,B):-right\_one(A,C),left\_two(C,B)$

我们首先通过设置不同数量的青蛙来扩展任务青蛙与荷叶问题 (见表 5). 尽管任务的程序空间巨大, 但程序本身很简单. 因此, 该解决方案不需要高阶的 MIL 系统. 仅使用一阶的 MIL 系统 Metagol 和 Metagol_F 来学习策略. 找到策略的学习时间如表 5 所示. 第 2 列表示实例 (作为 MIL 的实例集输入), 第 3 和第 4 列分别代表 Metagol 和 Metagol_F 的学习时间 (以 s 为单位). 显然, 改进后的系统 Metagol_F 能显著缩短该任务的学习时间. 此外, 实验结果表明 Metagol_F 始终会学习到和 Metagol 相同的策略.

表 5 青蛙与荷叶问题中 Metagol 和 Metagol_F 的学习时间 (s)

No.	实例集	Metagol	Metagol_F
1	$f([1,0,-1],[-1,0,1])$	0.033	0.006
2	$f([1,1,0,-1,-1],[-1,-1,0,1,1])$	0.128	0.056
3	$f([1,1,1,0,-1,-1,-1],[-1,-1,-1,0,1,1,1])$	19.488	5.368
4	$f([1,1,1,1,0,-1,-1,-1,-1],[-1,-1,-1,-1,0,1,1,1,1])$	9 761.168	1 526.604

表 6 展示了 Metagol 及其优化版 Metagol_F 的效率, 这里使用了与表 5 相同的 4 个实例, 重点关注推理、原子、全局堆栈和子句垃圾收集. 当只有一个实例时, 两个模型的表现相似, 表明它们的基本效率相当. 然而, 随着示例数量的增加, Metagol_F 表现出更好的性能, 推理次数更少、原子消耗更低, 表明其计算效率有所提高. 此外, 在 Prolog

的默认堆栈限制 (256 MB) 下, Metagol_F 需要更少的垃圾收集操作来回收未使用的内存. 即便在青蛙与荷叶问题中, Metagol_F 的推理次数相比 Metagol 也大大减少了, 大约减少了 98.65%.

表 6　青蛙与荷叶问题中 Metagol 与 Metagol_F 的性能与内存消耗对比

No.	Metagol			Metagol_F		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	4.62×10^5	6 164	3	4.62×10^5	6 164	3
2	3.53×10^6	6 165	5	3.90×10^5	5 989	3
3	2.90×10^8	6 166	5	7.82×10^6	5 990	4
4	8.19×10^{10}	6 167	6	1.10×10^9	5 991	5

5.2　机器人服务员

机器人服务员的任务是根据杯子上的标签将茶或咖啡倒入桌子上的空杯中. 我们用 $robot(A,B)$ 表示每个例子, 其中 A 和 B 分别表示机器人的初始状态和杯子的最终状态. 实验的输入设置包括背景知识、元规则集和例子集.

与文献 [21] 中的实验类似, 高阶系统 Metagol_AI 和 $\text{Metagol}_\text{AI\_F}$ 的背景知识包含两个部分, 即编译的背景知识和解释的背景知识. 而一阶系统 Metagol 和 Metagol_F 的背景知识仅包含编译的背景知识. 同时, 除了在一阶系统中使用的一阶元规则外, 高阶系统中的元规则集还包含高阶元规则.

我们随机生成 5 个不同规模的训练集, 大小从 1–5 不等, 并将时间限制设为 600 s. 在 4 个系统上重复进行 10 次实验后, 我们得到每个系统的平均学习时间.

图 6 展示了一阶和高阶 MIL 系统的学习结果. x 轴表示实例的数量. 在图 6(a) 中, y 轴表示一阶 MIL 系统 Metagol 和 Metagol_F 的平均学习时间. 在图 6(b) 中, y 轴表示 Metagol_AI 和 $\text{Metagol}_\text{AI\_F}$ 的平均学习时间. 原始系统 (Metagol 和 Metagol_AI) 和修改系统 (Metagol_F 和 $\text{Metagol}_\text{AI\_F}$) 的平均学习时间分别以红色和蓝色显示. 结果表明 Metagol_F 比 Metagol 学习速度更快, $\text{Metagol}_\text{AI\_F}$ 比 Metagol_AI 学习速度显著更快. 在最佳情况下, Metagol_F 的学习时间约为 Metagol 的 47.1%. 在最佳情况下, $\text{Metagol}_\text{AI\_F}$ 的学习时间约为的 30.26%.

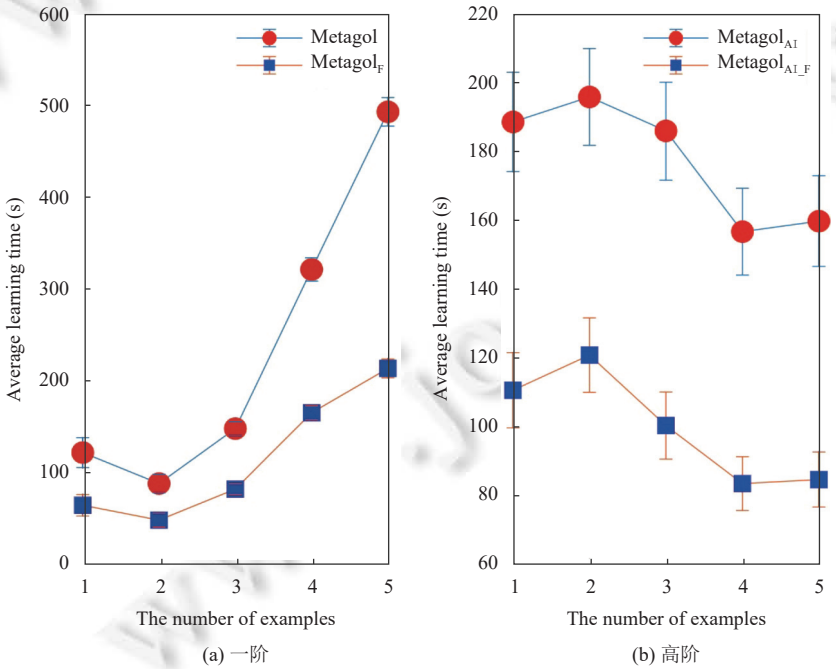


图 6　机器人服务员任务中一阶和高阶 MIL 系统的平均学习时间

表 7 和表 8 提供了一阶 MIL 系统 Metagol、$\text{Metagol}_{\text{F}}$ 以及高阶 MIL 系统 $\text{Metagol}_{\text{AI}}$、$\text{Metagol}_{\text{AI\_F}}$ 在执行机器人服务员任务期间内存消耗的对比分析. 分析表明, 优化版本 $\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 在推理次数上显著减少, 从而提高了计算效率. 对于一阶系统, $\text{Metagol}_{\text{F}}$ 相较于 Metagol 实现了约 44.18% 的平均推理次数减少. 同样, 在高阶系统中, $\text{Metagol}_{\text{AI\_F}}$ 相较于 $\text{Metagol}_{\text{AI}}$ 实现了约 42.80% 的平均推理次数减少. 这些显著的改进突显了优化系统中剪枝技术的有效性, 展示了其在处理具有更高计算效率、较低原子数和较少子句垃圾收集 (CGC) 能力的任务方面的优势, 且此改进在 Prolog 默认 256 MB 堆栈限制下得以实现.

表 7　在机器人服务员任务中, 一阶 MIL 系统 Metagol 与 $\text{Metagol}_{\text{F}}$ 之间的内存成本比较

Example	Metagol			$\text{Metagol}_{\text{F}}$		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	1.64×10^9	6373.33	7.25	1.10×10^9	6360.33	6.50
2	3.22×10^9	6359.67	7.67	1.60×10^9	6340.33	6.50
3	4.49×10^9	6331.33	8.33	2.13×10^9	6320.33	7.00
4	3.42×10^9	6388.33	9.33	2.30×10^9	6370.67	8.00
5	4.45×10^9	6388.33	9.00	2.12×10^9	6375.67	8.50

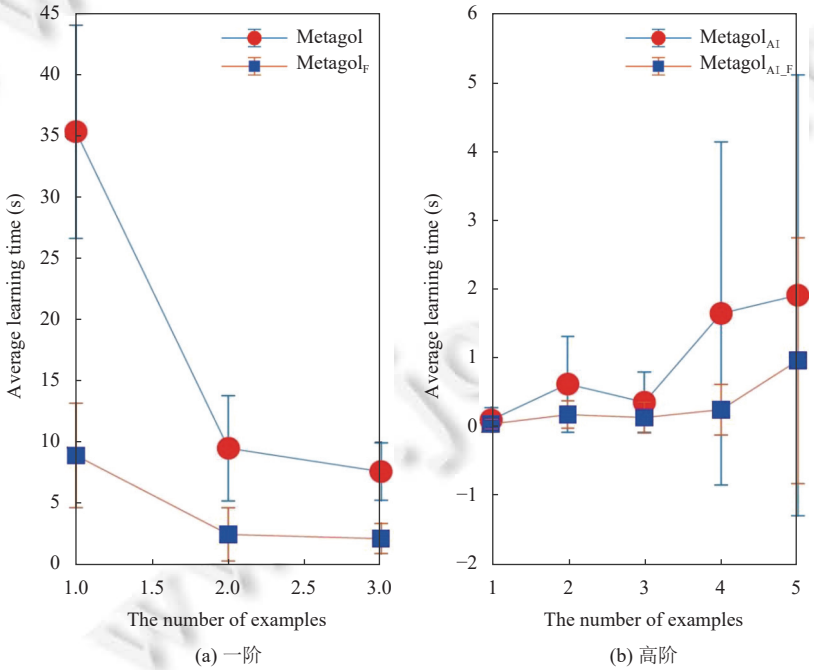
表 8　在机器人服务员任务中, 高阶 MIL 系统 $\text{Metagol}_{\text{AI}}$ 与 $\text{Metagol}_{\text{AI\_F}}$ 之间的内存成本比较

Example	$\text{Metagol}_{\text{AI}}$			$\text{Metagol}_{\text{AI\_F}}$		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	1.81×10^9	6444.29	8.29	1.07×10^9	6350.67	7.00
2	2.74×10^9	6444.67	8.67	1.89×10^9	6340.33	7.50
3	4.54×10^9	6445.00	9.00	2.23×10^9	6320.33	8.33
4	4.89×10^9	6444.67	10.00	2.90×10^9	6370.67	9.00
5	4.65×10^9	6445.00	10.00	2.30×10^9	6375.33	8.50

5.3　象棋策略

本实验的设置与文献 [21] 相同. 此学习任务旨在学习一个简单的国际象棋策略, 即维持一道黑卒的防线以实现升变. 在初始状态下, 卒子被放置在不同的横行上, 而在最终状态下, 它们位于第 8 行, 这里忽略了白方的干扰. 我们随机生成了 5 组不同大小的训练集, 大小范围为 1–5, 设定时间限制为 600 s, 然后使用上述 4 个系统学习这个国际象棋策略. 经过 10 次实验后, 我们得到了结果.

图 7 展示了 4 个 MIL 系统在国际象棋策略任务上的结果. 很明显, 改进的 MIL 系统相较于原始的 MIL 系统可以显著减少学习时间. 除了在样例数为 4 和 5 时的一阶系统外, 算法 1 的效果没有显现, 因为它们都是超时情况.



(a) 一阶　(b) 高阶

图 7　象棋策略任务中一阶和高阶 MIL 系统的平均学习时间

表 9 和表 10 提供了对于象棋任务的内存使用分析, 比较了原始 MIL 系统 Metagol、$\text{Metagol}_{\text{AI}}$ 与它们的剪枝版本 $\text{Metagol}_{\text{F}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 的性能. 结果表明, 剪枝版本在显著减少推理次数上取得了成功. 具体来说, $\text{Metagol}_{\text{F}}$ 平均推理次数减少了大约 15.88%, 而 $\text{Metagol}_{\text{AI\_F}}$ 的减少约为 12.77%, 相较于它们各自的基础系统. 剪枝后的 MIL 系统在 Prolog 默认堆栈限制内实现了更少的原子数量和减少的子句垃圾收集 (CGC).

表 9 在象棋策略任务中 Metagol 和 $\text{Metagol}_{\text{F}}$ 的内存消耗比较

Example	Metagol			$\text{Metagol}_{\text{F}}$		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	5.79×10^6	6428.00	11.00	5.07×10^6	6258.67	6.00
2	8.14×10^6	6430.00	14.00	6.38×10^6	6260.00	10.00
3	8.98×10^6	6429.33	15.00	7.76×10^6	6429.33	13.00

表 10 在象棋策略任务中 $\text{Metagol}_{\text{AI}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 的内存消耗比较

Example	$\text{Metagol}_{\text{AI}}$			$\text{Metagol}_{\text{AI\_F}}$		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	3.85×10^6	6429.10	12.40	3.46×10^6	6420.33	11.00
2	1.22×10^7	6429.80	15.10	1.10×10^7	6425.00	14.00
3	8.84×10^6	6430.00	15.60	7.95×10^6	6426.67	14.50
4	2.34×10^7	6429.78	15.11	2.10×10^7	6427.00	14.00
5	4.46×10^7	6429.89	15.33	3.41×10^7	6428.67	14.00

5.4 删除列表尾元素

删除列表尾元素的目标是学习一个程序 *droplasts*, 它能从给定列表的每个子列表中删除最后一个元素. 在此实验中, 我们随机生成了 5 个不同大小的训练集, 范围为 1–5, 并将时间限制设为 600 s.

由于一阶程序的表达能力有限, 一阶 MIL 系统 (Metagol 和 $\text{Metagol}_{\text{F}}$) 在此任务中均失败. 图 8 显示了两个高阶 MIL 系统 $\text{Metagol}_{\text{AI}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 的结果. 再一次, $\text{Metagol}_{\text{AI\_F}}$ 成功提高了学习效率. 同时, 高阶系统的平均学习时间被缩短了大约 1/4, 并且它们学习到了相同的程序. 在最佳情况下, $\text{Metagol}_{\text{AI\_F}}$ 的学习时间约为 $\text{Metagol}_{\text{AI}}$ 的 63.79%.

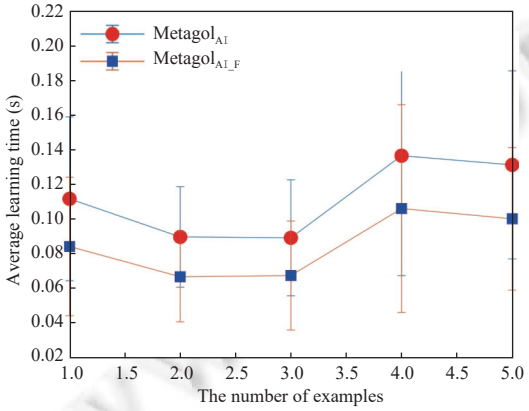


图 8 删除列表尾元素任务中高阶 MIL 系统 $\text{Metagol}_{\text{AI}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 的平均学习时间

表 11 对比了 $\text{Metagol}_{\text{AI}}$ 和 $\text{Metagol}_{\text{AI\_F}}$ 在解决删除列表尾元素问题上的内存成本. 值得注意的是, 尽管全局堆栈约束相同, $\text{Metagol}_{\text{AI\_F}}$ 在减少推理和优化垃圾收集方面表现出了显著改进. 这种效率指向了 $\text{Metagol}_{\text{AI\_F}}$ 在内存管理和计算优化方面的优越性能, 突显了其在处理复杂任务时的更高资源效率. 表 11 展示了 $\text{Metagol}_{\text{AI}}$ 和

$Metagol_{AI\_F}$ 在 *droplasts* 问题上的内存成本对比. 显著地, $Metagol_{AI\_F}$ 在全局堆栈限制下推理大幅减少, 平均减幅约为 46.12%. 与 $Metagol_{AI}$ 相比, $Metagol_{AI\_F}$ 需要更少的子句垃圾收集.

表 11 删除列表尾元素任务中系统 $Metagol_{AI}$ 和 $Metagol_{AI\_F}$ 之间的内存成本比较

Example	$Metagol_{AI}$			$Metagol_{AI\_F}$		
	Inferences	Atoms	CGC	Inferences	Atoms	CGC
1	3.17×10^6	6400.90	15.10	1.74×10^6	6401.00	12.33
2	3.53×10^6	6401.00	18.10	1.83×10^6	6401.00	16.00
3	3.63×10^6	6401.00	16.90	1.82×10^6	6401.00	15.33
4	3.77×10^6	6401.00	16.40	1.91×10^6	6401.00	14.67
5	3.57×10^6	6400.90	17.50	2.22×10^6	6401.00	15.33

6 相关工作

6.1 归纳逻辑程序设计相关工作

ILP 是一个专注于从例子和背景知识中归纳构建一阶子句理论的学科. 一阶逻辑归纳理论的开创性工作始于 20 世纪 70 年代. 作为著名的 ILP 系统之一, 基于相对最不一般化的 Golem 由 Muggleton 等人[23]提出. 其他依赖于一阶逻辑归纳的方法也相继被提出. 例如, Duce 是一个由 Muggleton[24]提出的基于命题 Horn 子句归纳的机器学习系统. Muggleton 等人[22]提出了一个系统 Cigol, 通过发明一阶 Horn 子句谓词自动补全背景知识中的不完整信息. 继此工作之后, Rouveirol 等人[25]将 Cigol 中引入的操作符扩展到非单位子句. Linus[26]是一个基于命题属性-值语言的方法. CLINT[27]是一个基于主动学习的 ILP 系统, 而 Progol[4]则是基于模式导向逆蕴涵的系统.

6.2 元解释学习相关工作

由 Muggleton 等人[2]于 2014 年开发的 MIL 框架最初应用于归纳语法推断, 随后在 2015 年扩展到学习高阶二元 Datalog 程序[15], 并进一步扩展到从少量实例进行递归数据转换, 涵盖半结构化和非结构化数据[28]. Cropper 等人[8]引入了蕴涵约简算法和推导约简算法[9]以减少假设空间. 随后的研究[29]提出了一种使用背景知识预先识别假设约束的方法, 大大减少了学习时间. 与这些在学习过程之前应用算法的方法相比, 本文提出的方法在学习阶段通过记录和检查失败的 Goals 来应用剪枝算法, 以避免重复证明.

为了学习更好的程序, 现有的 MIL 系统优化要么最小化程序的资源复杂性[30], 要么减少学习程序的文本复杂性[21]. Metaopt[19]扩展了 Metagol 以支持学习高效的程序. 它在假设搜索过程中维护一个成本, 并利用该成本来修剪假设空间. 与这些工作相比, 我们的优化策略主要集中于解决 MIL 学习过程中重复证明的问题, 以提高学习效率. 值得注意的是, 由于上述工作基于 Metagol 系统, 重复证明的问题在这些系统中也存在. 从理论上讲, 本文提出的算法也可以应用于这些 MIL 系统.

Kazmi 等人[31]扩展了一种基于答案集编程 (ASP) 的 ILP 方法 (名为 XHAIL), 在假设泛化算法中引入了修剪机制. 他们的工作使用了一种优化方法, 并允许使用次优结果. 其改进重点在于在保持相似质量的同时对更大数据集进行学习. 与此不同, 我们的优化目标是用更少的时间学习完全相同的程序. Kaminski 等人[32]指出, 直接的 MIL 编码将导致巨大规模的底层程序和搜索空间. 他们的工作使用答案集编程 (ASP) 通过在 ASP 的 HEX 扩展中编码 MIL 来解决 MIL 问题, 利用 HEX 接口机制缓解了底层问题. 相比之下, 我们的方法专注于确保底层和非底层目标不被多次证明.

Cropper 等人[33]提出了一种结合答案集编程和 Prolog 编程的 ILP 系统 Popper, 该系统通过“从失败中学习”策略逐步缩小假设空间, 从失败的假设中提取约束以提升学习效率. Popper 基于 ASP, 避免了 Prolog 回溯搜索的冗余问题, 但在高复杂度程序生成方面仍存在一定局限性. 本文的剪枝算法则专为元解释学习设计, 以减少 MIL 中的冗余证明, 提升其在复杂任务中的学习效率. Wang 等人[34]提出了一种改进的元解释学习方法, 称为 MILER, 该方法通过向背景知识中扩展谓词来引入可复用的程序模式以解决 MIL 在庞大程序搜索空间中的效率限制问题.

与之相比, 本文直接优化 MIL 算法, 通过剪枝减少冗余证明, 以提升复杂任务中的执行效率. 表 12 简要比较了 Aleph、Progol、Metagol、Popper 和 XHAIL 等常见 ILP 系统的核心特性.

表 12 一些 ILP 系统的简化比较

特性	Aleph	Progol	Metagol	Popper	XHAIL
假设	一般	确定性	确定性	确定性	确定性
语言偏误	模型	Datalog	元规则	声明	声明
谓词构造	否	否	是	否	否
递归	部分	是	是	是	是
最优	否	否	是	是	否
假设约束	是	是	否	是	是

7 结论与未来工作

本文揭示了元解释学习的一个固有缺陷, 即冗余证明. 在此基础上, 引入了一种基于记忆的裁剪算法来裁剪 MIL 中的冗余证明, 以提高学习效率, 包括缩短学习时间和生成完全相同的程序. 我们还提供了算法的实验评估和理论分析. 从实验结果可以看出, 程序空间越大, 学习时间减少越显著. 本文的观察结果揭示了在较大程序搜索空间应用裁剪算法时, 其有效性的显著提升. 这种改进在相同堆栈限制条件下表现为两个重要方面. 首先, 程序搜索空间的增加会导致学习时间的显著减少. 其次, 由于一个旨在最小化无效推理过程的失败数据库的直接影响, 冗余推理的次数显著减少. 尽管基于记忆的裁剪算法需要额外的内存来保存失败信息, 但由于现代高级计算机资源的存在, 几乎没有带来负面影响. 在我们裁剪算法的理论分析中, 证明了在内存无限可用的假设下程序搜索空间的减少率. 然而, 当将理论期望与实际程序学习时间的减少进行比较时, 发现了由于两个主要因素导致的差异. 首先, Prolog 操作机制的实际限制, 尤其是堆栈限制, 违背了我们关于无限内存可用性的理论假设. 这种限制本质上限制了在学习过程中可以有效管理的搜索空间大小. 其次, 裁剪算法的实现涉及大量计算, 因为需要识别失败目标以及验证和更新数据库的操作. 这些活动会带来额外的时间成本, 而这些在理论模型中未被考虑. 总体而言, 内存成本对本文裁剪算法的影响是可以接受的. 在我们的实验中, 裁剪算法的额外内存使用没有引起任何“内存不足”错误, 相反, 在机器人服务员任务的一阶和高阶实验中, 分别有 12% 和 24% 的超时案例通过提出的裁剪算法得以解决.

在不久的将来, 我们将专注于几个领域以进一步优化所提算法. 首先, 将研究更高效的数据结构和存储方法, 以提高算法的性能和计算效率. 这可能需要使用更高级的编程工具和框架来优化内存管理和数据组织. 其次, 计划探索将元解释学习与其他机器学习技术 (如神经网络) 结合, 以实现更复杂和复杂的程序合成任务. 我们预计这些措施不仅能优化现有算法, 还能显著扩展其实际适用性.

References:

[1] Muggleton S, De Raedt L. Inductive logic programming: Theory and methods. The Journal of Logic Programming, 1994, 19–20: 629–679. [doi: 10.1016/0743-1066(94)90035-3]

[2] Muggleton SH, Lin DH, Pahlavi N, Tamaddoni-Nezhad A. Meta-interpretive learning: Application to grammatical inference. Machine Learning, 2014, 94(1): 25–49. [doi: 10.1007/s10994-013-5358-3]

[3] Quinlan JR, Cameron-Jones RM. FOIL: A midterm report. In: Proc. of the 1993 European Conf. Machine Learning. Vienna: Springer, 1993. 1–20. [doi: 10.1007/3-540-56602-3_124]

[4] Muggleton S. Inverse entailment and Progol. New Generation Computing, 1995, 13(3–4): 245–286. [doi: 10.1007/bf03037227]

[5] Blockeel H, De Raedt L. Top-down induction of first-order logical decision trees. Artificial Intelligence, 1998, 101(1–2): 285–297. [doi: 10.1016/s0004-3702(98)00034-4]

[6] Srinivasan A. The Aleph Manual. 2001. https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html

[7] Lin DH, Dechter E, Ellis K, *et al*. Bias reformulation for one-shot function induction. In: Proc. of the 21st European Conf. on Artificial Intelligence. Prague: IOS Press, 2014. 525–530.

[8] Cropper A, Muggleton SH. Logical minimisation of meta-rules within meta-interpretive learning. In: Proc. of the 24th Int'l Conf. on Inductive Logic Programming. Nancy: Springer, 2015. 62–75. [doi: 10.1007/978-3-319-23708-4_5]

[9] Cropper A, Tourret S. Derivation reduction of metarules in meta-interpretive learning. In: Proc. of 28th Int'l Conf. on Inductive Logic Programming. Ferrara: Springer, 2018. 1–21. [doi: 10.1007/978-3-319-99960-9_1]

[10] Morel R, Cropper A, Ong CHL. Typed meta-interpretive learning of logic programs. In: Proc. of the 16th European Conf. on Logics in Artificial Intelligence. Rende: Springer, 2019. 198–213. [doi: 10.1007/978-3-030-19570-0_13]

[11] Hocquette C. Efficient instance and hypothesis space revision in meta-interpretive learning [Ph.D. Thesis]. London: Imperial College London, 2022. [doi: 10.25560/97356]

[12] Cyrus D, Milani GA, Tamaddoni-Nezhad A. Explainable game strategy rule learning from video. In: Proc. of the 17th Int'l Rule Challenge and 7th Doctoral Consortium @ RuleML+RR 2023 co-located with 19th Reasoning Web Summer School (RW 2023) and 15th DecisionCAMP 2023 as part of Declarative AI 2023. 2023.

[13] Cyrus D, Trewern J, Tamaddoni-Nezhad A. Meta-interpretive learning from fractal images. In: Proc. of the 32nd Int'l Conf. on Inductive Logic Programming. Bari: Springer, 2023. 166–174. [doi: 10.1007/978-3-031-49299-0_12]

[14] Muggleton SH, Santos JCA, Tamaddoni-Nezhad A. TopLog: ILP using a logic program declarative bias. In: Proc. of the 24th Int'l Conf. on Logic Programming. Springer, 2008. 687–692. [doi: 10.1007/978-3-540-89982-2_58]

[15] Muggleton SH, Lin DH, Tamaddoni-Nezhad A. Meta-interpretive learning of higher-order dyadic Datalog: Predicate invention revisited. Machine Learning, 2015, 100(1): 49–73. [doi: 10.1007/s10994-014-5471-y]

[16] Tourret S, Cropper A. SLD-resolution reduction of second-order Horn fragments. In: Proc. of the 16th European Conf. on Logics in Artificial Intelligence. Rende: Springer, 259–276. [doi: 10.1007/978-3-030-19570-0_17]

[17] Gallier JH. Logic for Computer Science: Foundations of Automatic Theorem Proving. 2nd ed., New York: Dover Publications Inc., 2015.

[18] Nienhuys-Cheng SH, Wolf R. Foundations of Inductive Logic Programming. Berlin: Springer, 1997. [doi: 10.1007/3-540-62927-0]

[19] Cropper A, Muggleton SH. Learning efficient logic programs. Machine Learning, 2019, 108(7): 1063–1083. [doi: 10.1007/s10994-018-5712-6]

[20] Cropper A, Morel R, Muggleton SH. Learning higher-order programs through predicate invention. In: Proc. of the 34th AAAI Conf. on Artificial Intelligence. New York: AAAI Press, 2020. 13655–13658. [doi: 10.1609/aaai.v34i09.7113]

[21] Cropper A, Muggleton SH. Learning higher-order logic programs through abstraction and invention. In: Proc. of the 25th Int'l Joint Conf. on Artificial Intelligence. New York: AAAI Press, 2016.

[22] Muggleton S, Buntine W. Machine invention of first-order predicates by inverting resolution. In: Proc. of the 5th Int'l Conf. on Machine Learning. Ann Arbor: Morgan Kaufmann, 1988. 339–352. [doi: 10.1016/b978-0-934613-64-4.50040-2]

[23] Muggleton S, Feng C. Efficient induction of logic programs. In: Proc. of the 1990 Int'l Conf. on Algorithmic Learning Theory. Tokyo: Springer, 1990. 368–381.

[24] Muggleton S. Duce, an Oracle-based approach to constructive induction. In: Proc. of the 10th Int'l Joint Conf. on Artificial Intelligence. Milan: Morgan Kaufmann Publishers Inc., 1987. 287–292.

[25] Rouveirol C, Puget JF. Beyond inversion of resolution. In: Proc. of the 7th Int'l Conf. Austin: Morgan Kaufmann, 1990. 122–130. [doi: 10.1016/b978-1-55860-141-3.50018-3]

[26] Lavrač N, Džeroski S, Grobelnik M. Learning nonrecursive definitions of relations with Linus. In: Proc. of the 1991 European Working Session on Learning on Machine Learning. Porto: Springer, 1991. 265–281. [doi: 10.1007/bfb0017020]

[27] De Raedt L, Bruynooghe M. CLINT: A multi-strategy interactive concept-learner and theory revision system. In: Proc. of the 1st Int'l Workshop on Multi-strategy Learning Workshop. Virginia, 1991. 175–191.

[28] Cropper A, Tamaddoni-Nezhad A, Muggleton SH. Meta-interpretive learning of data transformation programs. In: Proc. of the 25th Int'l Conf. on Inductive Logic Programming. Kyoto: Springer, 2016. 46–59. [doi: 10.1007/978-3-319-40566-7_4]

[29] Cropper A, Hocquette C. Learning logic programs by discovering where not to search. In: Proc. of the 37th AAAI Conf. on Artificial Intelligence. Washington: AAAI Press, 2023. 6289–6296. [doi: 10.1609/aaai.v37i5.25774]

[30] Cropper A, Muggleton SH. Learning efficient logical robot strategies involving composable objects. In: Proc. of the 24th Int'l Conf. on Artificial Intelligence. Buenos Aires: AAAI Press, 2015. 3423–3429.

[31] Kazmi M, Schüller P, Saygın Y. Improving scalability of inductive logic programming via pruning and best-effort optimisation. Expert Systems with Applications, 2017, 87: 291–303. [doi: 10.1016/j.eswa.2017.06.013]

[32] Kaminski T, Eiter T, Inoue K. Meta-interpretive learning using HEX-programs. In: Proc. of the 28th Int'l Joint Conf. on Artificial

Intelligence. Macao: AAAI Press, 2019. 6186–6190.

[33] Cropper A, Morel R. Learning programs by learning from failures. Machine Learning, 2021, 110(4): 801–856. [doi: 10.1007/s10994-020-05934-z]

[34] Wang R, Sun J, Tian C, Duan ZH. Meta-interpretive learning with reuse. Mathematics, 2024, 12(6): 916. [doi: 10.3390/math12060916]

王榕(1991－), 女, 博士, 主要研究领域为归纳逻辑设计, 机器学习.

于斌(1990－), 男, 博士, 讲师, CCF 高级会员, 主要研究领域为软件系统的形式验证, 区块链, 智能合约.

田聪(1981－), 女, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为模型检测理论, 时序逻辑和自动机, 软件系统的形式验证, 软件工程.

段振华(1948－), 男, 博士, 教授, CCF 会士, 主要研究领域为模型检测, 时序逻辑, 软件系统的形式验证, 时序逻辑编程.

孙军(1978－), 男, 博士, 教授, 主要研究领域为软件工程, 编程, 安全软件工程, 程序分析, 形式化方法.