arXiv:2603.05258v1 [cs.LO] 5 Mar 2026

非合流证明搜索的约束学习

Michael Rawson[1(✉)] Clemens Eisenhofer[2], 和 Laura Kovács[2]

[1] 南安普顿大学，南安普顿，英国
michael@rawsons.uk[2] 维也纳工业大学，维也纳，奥地利
{clemens.eisenhofer,laura.kovacs}@tuwien.ac.at

摘要。 非合流表格演算（如连接 tableau 演算系统）中的证明搜索会遭遇过度回溯，但对回溯施加简单限制又会导致不完备。我们采用约束学习来减少经典一阶连接演算中的回溯，同时保持完备性。我们迭代改进一种用于连接驱动搜索的初始约束学习语言，从而在实践中大幅减少回溯。该方法或可用于其他非合流表 tableau 演算系统中的证明搜索。

关键词： 约束学习 · 连接表 · 回跳

1 引言

自动定理证明的最先进方法基于穷尽搜索，使用证明演算来探索可能证明的空间。证明搜索可以采用回溯或非回溯方式。当底层演算因搜索过程中先前所作的选择而陷入“僵局”时，就需要回溯搜索。必须撤销这些先前的选择，并作出其他选择，搜索才能继续；我们将这一过程称为回溯。

有些演算不需要回溯，例如合流表格演算。超位置和实例生成等演算也属于这一类。无回溯演算有时更受青睐，而且通常具有理论优势。然而，在某些情况下，非合流演算在实践中更有效，或者因其他原因而更受青睐。因此，我们关注改进回溯式证明搜索的表现。

回溯搜索本身并非问题：最先进的 SAT 求解器都统一基于回溯过程。问题出在回溯行为异常时：回溯得太少，并且在死路的根本原因没有改变时，反复尝试关闭相同的目标。顺便一提，回溯过多 也会造成问题。例如，在 SMT 求解中，向各自的判定过程中添加或移除理论文字是一项相对昂贵的操作，应尽可能避免，因此更倾向于采用较小的回溯步长。

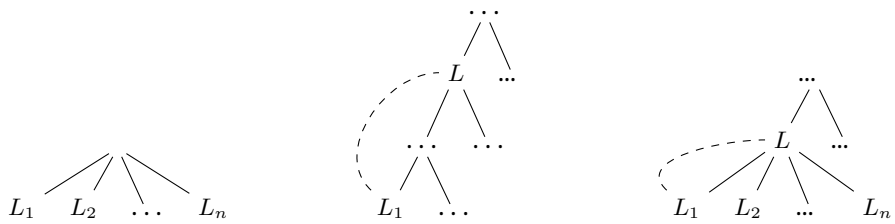


图 1。 子句连接表演算的三条推理规则：起始、归约和扩展。在起始 规则和扩展 规则中，$C = L_1 \vee L_2 \vee \ldots \vee L_n$ 是输入集中的一个子句，其变量已重命名，以与语义树中的变量相互区分。在归约 规则和扩展 规则中，我们要求 $\sigma(\neg L) = \sigma(L_1)$，即 L 和 L_1 相连接（以虚线表示）。

我们通过借鉴约束满足领域的一种名为约束学习[3] 的技术来解决这一问题。在寻找封闭语义树的过程中，我们有时会陷入死路，此时表格演算中已没有可应用的推理。此时，我们会分析为何没有推理可用，并学习 一个约束子句，以防止我们再次得到因同样原因而陷入僵局的类似语义树。不断累积的约束数据库有助于引导搜索转向更有希望的区域，或者最终证明不存在封闭语义树。

一个可能引起困惑的地方。 熟悉 SAT 求解、实例生成和/或反驳式定理证明的读者可能会怀疑，我们是在学习输入问题的推论；如果我们推导出一个显然不可满足的约束，例如空子句或 $0 = 1$，那么输入问题就是不可满足的。事实并非如此：我们学习的是有关搜索空间的约束，这类约束表明，在（给定的资源界限下）找不到封闭表板。

2 背景与动机

我们假定读者已熟悉表方法 [10] 以及经典一阶逻辑。连接方法 [6], 、连接 tableau 演算 [21], 、模型消去 [22], 与配对方法 [2] 是密切相关的证明搜索方法。鉴于听众，我们将采用连接表的语言来呈现我们的工作。连接细化要求对开放分支的任何添加都与叶文字相连 （即包含一个具有相反极性的文字），从而产生非常强的目标导向效应。其代价是即使对于命题逻辑也会失去汇合性：选择错误的语义树扩展可能无法闭合该语义树。

[3] 在布尔可满足性领域中更广为人知的名称是子句学习

通过将命题连接表提升为带有全局替换 σ的自由变量表格演算，可以得到一个针对经典一阶逻辑 [21]的完备演算。我们在图1中给出子句连接表演算的三条推理规则：起始 向空语义树添加一个子句，归约通过将叶文字连接到其路径上的某个文字来闭合一个分支，扩展则添加一个与开放分支相连的子句。只需稍作修改，连接方法便可适用于其他逻辑，例如直觉主义 [26]或模态 [28] 逻辑，或用于非子句证明搜索 [29]。

该演算的简洁性使得定理证明器通常可以非常紧凑，且常常利用 Prolog 及相关技术 [26,31,38]。尽管语义树方法已不再是经典一阶定理证明的最先进方法，但在存在大量无关公理的情况下（这是交互式定理证明器的一个关键应用 [23]），或在专门化设置中 [41]，它们在证明猜想方面仍具有竞争力。它们还在将机器学习应用于定理证明的实验中 [20,36,42], 找到了新的用武之地，因为其简洁性以及在一定程度上的回溯特性，使它们成为颇具吸引力的选择。这里我们假设等词已从输入中预处理移除 [25,8,33]„ 尽管可以扩展连接演算以支持等词推理 [6,32,4]。

2.1 连接驱动搜索中的过度回溯

为了保持完备性，命题连接系统必须考虑向语义树中添加内容的其他可能方式，但一旦某个分支被闭合，它就可以保持闭合。在一阶层面上，还必须考虑闭合同一分支的其他方式：这是因为闭合一个分支可能会以某种方式绑定全局替换中的变量，从而阻止之后闭合另一个分支。

Otten 注意到，这一要求在某些情况下会产生大量回溯 [27]。他在 leanCoP 的搜索例程中引入了 Prolog 割 ，使其不完备，但显著减少了回溯，并在许多情况下提高了性能。后来，Färber 对这些割的行为进行了广泛研究，并开发了若干变体 [14]。Färber 的所有变体都不完备，但其中一些在实践中比其他变体有效得多。我们将在第 6 节中使用他的 meanCoP 系统作为比较基准。

2.2 术语与约定

我们会非正式地使用一些术语，希望这有助于理解，而非造成困惑。由于基于语义树的一阶定理证明器通常会同时操作一棵语义树和一个替换，在不会引起混淆时，我们将二者统称为“语义树”。如果尝试应用某个演算推理却未能成功，我们就说它失败了。此外，如果我们能够判断一棵语义树绝不可能闭合，就说它停滞了。失败推理和停滞语义树都可以用约束来解释 ，我们将这种约束称为原因。

我们需要明确指明语义树中的特定位置。我们采用一种显而易见的表示法：对于任意位置 p， $p.i$ 是 p下方 i^{th} 分支上的位置。空位置表示语义树的根。语义树中的一阶变量根据其子句所附着位置的位置 命名，因此在回溯过程中保持一致 。对于一阶变量，我们使用u、 v、 w、 x、 y和 z；对于常量，使用 c和 d；对于函数，使用 f；对于谓词，使用 P、 Q、 R和 S 。最后，我们省略项和文字中的括号，并将 $\neg\neg L$ 视为与文字 L相同。

2.3 约束学习

约束学习 [11] 是约束满足与人工智能中一种广为人知、但定义较为模糊的方法。就我们的目的而言，既不必正式定义约束学习，也无需探讨其全部发展，但其核心思想如下。在为一组约束寻找解的回溯搜索中，我们可能会遇到死路：此时沿任何可用方向迈出一步都会违反某条约束。通过理由提取过程，可以将这种情况归咎于搜索先前决策的一个子集。随后会向约束集中添加一条约束，强制理由中的元素不能被同时全部选中，从而防止搜索再次陷入类似的不利情形。这种学习得到的 约束（通常以冲突子句的形式出现）还可用于执行回跳：即回溯超过一层。

约束学习对布尔可满足性 [37]（SAT）求解尤为有效，并且是现代冲突驱动子句学习 （CDCL）SAT 求解器的基础，因而也是可满足性模理论（SMT）求解器 [5]的基础。

2.4 运行示例

我们在全文中使用一个具体示例。考虑以下一阶子句集合：

$$\forall xyz.\ Px \vee Qy \vee Rxy \vee Pz \tag{1}$$

$$\forall x.\ \neg Px \vee S \tag{2}$$

$$\neg S \vee \neg Pc \tag{3}$$

$$\neg Qd \tag{4}$$

$$\neg Pfc \tag{5}$$

$$\forall x.\ \neg Rxc \tag{6}$$

$$\forall x.\ \neg Rdx \tag{7}$$

图 2 展示了由这组子句构建的连接表。其中只有一个开放分支 Rxy，在位置 3 处以方框标出。当前替换为 $\{\ x \mapsto c,\ y \mapsto d,\ z \mapsto fc,\ w \mapsto x\ \}$。 Rxy唯一可用的扩展步骤是 $\neg Rdv$ 或 $\neg Rvc$。这个语义树陷入了僵局，因为这两种扩展都不可能进行。在构建的较早阶段，也就是 $x \mapsto c$ 和 $y \mapsto d$ 之前，两种扩展都可以进行。请注意，位置 4 处的 Pz 如何闭合并不重要，而位置 1 和 2 处的子表格则导致了这条死路。

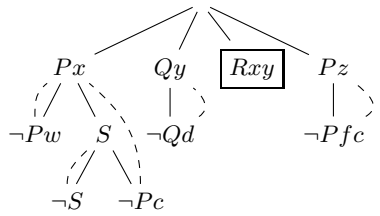


图 2。 运行示例：由子句 1–7 构建的连接表。

3 学习约束

我们提出在连接 tableau 演算的证明搜索过程中学习并存储约束，以避免因相似原因反复陷入死路。现在，我们将定义一种约束语言，用来说明在给定情形下为何无法进行任何推理步骤，从而能够在第 4 节设计一种改进的搜索过程。假设某个推理步骤通常适用于语义树中的一个开放分支。如果该步骤不适用，那么必定是语义树其他位置的某些规则应用阻止了它。因此，我们将约束语言定义为基于连接 tableau 演算的推理规则。我们将在第 5 节进一步完善这种语言，但现在先来看以下定义。

定义1（简化约束语言）。约束是原子集合。每个原子均为以下形式之一：

1. $\mathcal{S}_C$，表示使用子句 C 开始构建语义树；*2.* $\mathcal{R}_p^q$，表示从位置 p 归约到语义树中的祖先 q；*3.* $\mathcal{E}_{C/i}^p$，表示通过连接到子句 C 的 i^{th} 文字来扩展位置 p。

请注意，每个原子都包含应用该步骤的开放目标（或根）。这种语言足以解释为什么推理 j 在其他情况下本来是可能的，但当前在语义树中却无法进行，并以一种能够涵盖整类受类似影响的语义树的方式描述这种情况。具体做法是找出已经应用于语义树的推理步骤的一个子集，而该子集阻止了推理 j 的应用。

定义 2（推理失败的原因）。取语义树 T 中的一条开放分支 B，该语义树由一系列推理 I 构造而成。我们通过仅应用产生 B 所必需的那些推理 $I' \subseteq I$ 来构造一个子表 T'，即起始子句以及沿着通往 B 的路径进行的一系列扩展。假设存在一条推理 j，它可以应用于 T' 中的 B，但不能应用于 T 中。若额外将一组最小集合 $E \subseteq I \setminus I'$ 应用于 T'，从而阻止在所得语义树中的 B 处应用 j，那么这组集合就是无法在 T 中应用 j 的原因。

示例 *1*（推理失败的原因 *1*）。考虑图 2 中的语义树。要闭合它，必须扩展剩下的开放分支 Rxy。假设我们希望用子句 7 扩展该分支。起初这是可行的，但此时全局替换包含 $x \mapsto c$，因此无法进行扩展。

我们可以用自己的语言解释这一点：只需注意，使扩展不可能的先前推理的最小集合，就是那些在位置 1 关闭分支的推理 Px。位置 2 和 3 的另外两个分支无关紧要，尽管它们也已关闭，并且会影响全局替换。如果我们取先前推理的最小集合，就得到 $\{ \mathcal{S}_1, \mathcal{E}^1_{2/1}, \mathcal{E}^{1.2}_{3/1}, \mathcal{R}^1_{1.2.2} \}$

示例 *2*（推理失败的原因 *2*）。我们回到图 2 中的语义树，但现在考虑用单子句 6 扩展开放分支。同样，我们注意到，只有位置 2 的分支 Qy 与解释此次扩展为何失败有关，并得出原因 $\{ \mathcal{S}_1, \mathcal{E}^2_{4/1} \}$。

现在我们来解释为什么语义树会陷入停滞。这包括两部分：指出存在一个开放分支 B，并说明没有任何推理可以应用于 B。

定义 3（停滞语义树的理由）。取自定义 *2* 的 T、B、I、T' 和 I'，并令 J 为 T' 中 B 的可能推理集合。假设没有任何 $j \in J$ 可应用于 T 中的 B。我们按如下方式计算每个推理 j 的理由集合 R_j：

1. 如果演算阻止 j 在 T 中，我们根据定义 *2* 计算解释 R_j。*2.* 如果 j 可以应用，但会导出一个同样停滞的语义树，我们递归地计算该语义树的理由 R'，并令 $R_j = R' \setminus \{ j \}$。

I' 将 B 描述为一个开放分支，而所有推理失败理由R_j 的并集表明，没有推理可以应用于 B。我们定义

$$\bigcup_j R_j \cup I'$$

作为 T 停滞的理由。

示例 *3*（停滞语义树的理由）。再次考虑图 2 中的语义树及其开放分支。如果唯一可能的扩展是使用子句 6 和 7 进行扩展，那么该语义树就会停滞，因为这两个子句在此处都无法应用。使用示例 1 和 2 中的理由集合，并注意到 I' 只不过是 S_1，我们得到

$$\{ \mathcal{S}_1,\ \mathcal{E}^1_{2/1},\ \mathcal{E}^{1.2}_{3/1},\ \mathcal{R}^1_{1.2.2},\ \mathcal{E}^2_{4/1} \}$$

作为 T 陷入停滞的原因。

一般来说，对于任何给定的停滞语义树，原因都不唯一：既可能有多个开放分支，也可能有多个导致推理失败的原因。选择其中一个原因即可，但有些原因可能比其他原因更有力。

算法 1 用于查找闭合语义树的迭代搜索例程。

```
T ←空语义树
约束 ← ∅
轨迹 ← 空
重复
  成功 ← 错误
  B ← 选择_打开_分支(T)
  学习 ← explain(T, B)
  对于所有 可能的推理 j 在 B 中 T 执行
    如果不 应用(T, j) 则
      学习 ← 学习 ∪ 计算_推理(T, j)
    否则如果 存在一个冲突子句 C ∪ {j} 被 j 和轨迹违反那么
      学习 ← 学习 ∪ C
    else
      轨迹 ← j :: 轨迹
      成功 ← 真
      跳出
    结束如果
  结束 for
  如果 成功则
    当学习约束被违反时
      i ← 弹出(轨迹)
      撤销(i, T)
    结束 while
    记录_学习到的_子句（学习）
  结束如果
直到 T 已关闭或学习内容为空
```

4 使用学习到的约束进行搜索

上一节中，我们定义了约束语言，使停滞语义树能够得到充分解释。现在应重新设计搜索算法，以利用这些学习到的约束。我们实现的方案与 CDCL SAT 求解器、SMT 求解器或约束满足系统中的方案类似。除当前语义树外，我们还维护一条*原子轨迹*，其中记录了对当前语义树为真的原子。

搜索算法会反复应用规则、陷入停滞、学习一个约束，然后进行回溯。当语义树已关闭或学习到空约束时，算法终止。我们保持以下不变量：没有任何学习到的约束会被当前轨迹*违反*：如果某个约束的所有原子都包含在轨迹中，则该约束被违反。如果语义树为空，则选择一个起始子句。否则，在每次迭代中，选择语义树的一个开放分支进行归约或扩展。如果可以进行任何此类推理，就将其应用于语义树，并将描述该推理结果的相应原子添加到轨迹中。另一方面，某个推理 j 可能无法进行，原因可能是：

1. 因为该演算不允许这样做，或

2. 因为添加其对应的原子会违反一个学习子句。

在第一种情况下，失败推理的原因按定义 2 计算。否则，其失败原因就是它会违反的学习到的约束，减去 与 j 对应的原子（定义 3）。这样，当开放分支上的所有可能推理都失败时，系统会针对停滞的语义树学习一个约束，使得该约束中的所有原子都在轨迹上。为恢复不变量，系统会回溯，直到至少有一个被违反的原子不再位于轨迹上。

整体过程如算法 1 所示：compute_reason 计算定义 2 意义上的原因，而 explain 则按定义 3 为 B 计算 I'。这比通常用于寻找封闭连接表的过程复杂得多，尤其是那些通过“lean”方法嵌入 Prolog 的过程 [26]。不过，它在一个方面更简单：无需在回溯点记住备选推理；如果不借助 Prolog 现有的回溯机制来实现，这可能会相当复杂 [19]。

4.1 资源界限

连接系统通常通过对某一度量（例如最长分支的长度）进行迭代加深来搜索。最初设定一个较小的界限，然后系统在当前界限的约束下开始搜索。如果在某一迭代加深层级不存在语义树，则提高界限并再次尝试搜索。我们在此采用这一方法：我们的搜索算法寻找受某一最大分支长度限制的语义树；若不存在这样的语义树，它最终将通过学习空子句而终止。

在某一迭代加深层级学到的约束不能在下一层级复用，否则我们的方法会变得不完备。可以修改约束语言，以表达（不）依赖于深度限制的约束，但这至少出于两个原因而实用价值有限。首先，下一迭代加深层级的搜索空间往往远大于已穷尽的上一层级，因此复用与深度限制无关、来自上一层级的约束帮助不大。其次，在实践中，这类约束很少。

4.2 可靠性、终止性与完备性

我们证明算法 1 在任意固定的深度限制下都会终止，并以此证明完备性。可靠性显然成立，因为该例程是在一个已有的可靠演算中进行搜索。

引理 1（终止性）。固定一个深度限制。算法 *1* 会终止。

证明。首先请注意，在任意深度限制下，对于任意有限的输入子句集合，可能的语义树数量是有限的，且它们的规模都是有限的。由于所有语义树的规模都是有限的，该例程最终要么闭合

该语义树并立即终止，要么陷入停滞。陷入停滞时，该例程会学习一条约束；根据构造，当前轨迹违反了该约束，从而至少阻止再次到达这一特定语义树。由于在同一深度限制内约束永不被遗忘，且可能的语义树数量有限，终止性得以保证，因为求解器最终会学习到消除所有可能语义树的约束。□

引理 2. 在给定深度限制下，证明演算中可达的任何封闭语义树都不会违反学习到的约束。

证明。 对学习到的约束的推导进行归纳。假设封闭语义树 T^*违反了学习到的约束 C。根据定义， C是构造 T^*所需规则的子集。构造由 C生成的中间语义树 T。从 T朝着 T^*执行下一推理步骤 j。在定义 3 中， C 的依据是：从 T出发的所有推理（包括 j）都不可能，要么是因为演算禁止这样做（定义 2），要么是因为违反了另一个约束 C'。根据归纳假设，应用 j 不会违反任何此类 C'，因此 j 在演算中必定不合法，由此得到矛盾。□

定理 1（完备性）。如果在某个深度限制下存在封闭语义树，算法 *1* 就会找到它。

证明。 根据终止性和引理 2。 □

5 精化约束语言

第 3 节中引入的简单约束语言足以阻止某些相似语义树类，但它对特定语义树的针对性很强，无法阻止我们可能希望阻止的所有相似语义树。此外，这种语言使用起来也相当笨拙，实际应用中很难高效地计算推理失败的原因：参见第 6.2 节。

因此，我们将每个原子分解为多个更小的原子，分为两类：在语义树的位置放置文字，以及将变量绑定到项。不过，其他方面 搜索过程保持不变，每次推理时都将一个或多个这样的原子压入轨迹。约束不仅更易于实现，而且也更强大，因为它们阻止的并非仅是语义树的某个特定推导，而是任何包含特定文字和变量绑定的语义树。

定义 4（精化约束语言）。 约束仍然是原子的集合。不过，每个原子要么是

1. $L@p$，一个文字 L 被放置在位置 p*2.* $x \mapsto t$，一个变量 x 绑定到一个项 t，其中 t 本身也可能是一个变量。

连接 tableau 演算中的每个推理都可以表示为这些操作的某种组合。在起始规则和扩展规则中添加子句，是通过将其文字放置在相应位置来完成的。扩展规则和归约规则中的文字连接，则通过计算所需的变量绑定来应用。

示例 *4*（精化约束学习）。考虑图 2 中的语义树。我们将用新的约束语言解释它为何陷入停滞。无法用 $\neg Rvc$ 扩展 Rxy，因为 $y \mapsto d$ 位于轨迹上，这是由于 Qy 与 $\neg Qd$ 相连。类似地，无法用 $\neg Rdv$ 扩展 Rxy，因为 $x \mapsto c$。要完成这一解释，我们还必须说明为何一开始就需要关闭 Rxy，不过这很简单：Rxy@3。因此，最终的解释为

$$\{\ Rxy@3,\ x \mapsto c,\ y \mapsto d\ \}.$$

5.1 无连接原子

在所提出的语言中，解释为什么一个开放分支无法归约，可能会变得过于具体。

示例 *5*（解释归约失败）。考虑一个处于深度限制处的开放分支 $\neg Pc$，其路径文字为 Px、Qc、Rcd、S，且替换包含 $x \mapsto d$。假设从根到叶节点的位置为 $p_1 \dots p_5$。显然，$\neg Pc$ 无法归约。对于 Px，该约束包含一些有用信息：如果 x 没有绑定到 d，这个分支就可以归约。然而，对于所有其他路径文字，唯一有用的信息是它们无法与$\neg Pc$连接，但语言中没有这种表达，因此我们必须学习

$$\{\ \neg Pc@p_5,\ Px@p_1, x \mapsto d,\ Qc@p_2,\ Rcd@p_3,\ S@p_4\ \}.$$

这种情况在实践中经常发生，并且会不必要地将学习到的约束特化为特定的路径文字序列。为避免这一问题，我们引入一种新的原子 $p \not\sim q$，表示无论替换为何，两个位置 p 和 q之间都不可能建立连接。每当在位置 q将文字添加到语义树时，都会检查其路径，查看它可以与位置 p 处的哪些文字归约；若无法归约，则将 $p \not\sim q$ 添加到轨迹中。在上述示例中，学习到的约束将为

$$\{\ \neg Pc@p_5,\ Px@p_1,\ x \mapsto d,\ p_2 \not\sim p_5,\ p_3 \not\sim p_5,\ p_4 \not\sim p_5\ \}.$$

而且更为一般，因为它没有指定哪些文字位于 p_2、p_3 或 p_4。

5.2 不等式

经典精化，例如正则性和消除重言式，可以显著提升连接系统 [21]的性能。传统上，这些技术通过不等式来实现。只需添加形如 $s \neq t$的不等式原子，我们就能在约束语言中自然地支持这一点。当不等式被证伪时，可以将该不等式以及证伪它所需的变量绑定作为学习到的约束，从而触发回溯。

6 实现与实验验证

我们实现了一个原型系统 hopCoP[4] ，用于试验约束学习。迄今为止，该实现采用命令式编程——尽管我们怀疑通过 assert/1 [18] 或许可以实现一个精简的 Prolog 版本——并且在很大程度上借鉴了 Bare Metal Tableaux Prover [19] 和 meanCoP [14]中的实现技术。hopCoP 实现了子句连接表演算，不含割 [27] ，并从猜想派生出的子句开始。除约束学习这一显而易见的例外之外，如果设置 meanCoP 的标志 --conj --nopaths[5]，hopCoP 的搜索例程就与 meanCoP 的相似。meanCoP 还实现了一个引理机制 [21] ，而 hopCoP 目前尚未具备这一机制。在第 6.1 节和第 6.2 节中，我们重点介绍了实现中的两个方面，它们或许会令类似系统的实现者感兴趣。

6.1 约束管理与冲突检测

在搜索过程中会学习到大量约束，若时间限制足够长，数量可达数百万。不过，这似乎不像 SAT 求解等其他场景中那么突出，而且我们发现，尝试清理旧约束并无益处，因此 hopCoP 会保留它学习到的所有约束。

将原子添加到轨迹时，仍有必要高效地从这个大型集合中查找冲突。我们采用一种 1-watch 文字方案 [24]：无需使用 SAT 求解中常见的 2-watch 文字方案，因为所有原子的极性都相同，而且单位传播用处不大。向轨迹添加一个原子时，可能会发现多个冲突：值得尝试选择能使最终学习到的约束最小化的冲突。hopCoP 会贪心地选择为迄今学习到的约束增加原子数最少的冲突。

6.2 计算解释

要解释为什么连接两个文字的推理无法进行，必须计算当前替换的一个子集，使它能够阻止这两个文字合一。无疑，可以构造复杂的变量跟踪方案来快速完成此操作，但对于我们的应用而言，以下过程的速度已足够快。

1. 使用新的临时替换 τ 对这两个文字进行合一。由于使用空替换时可以进行该推理（但使用 σ 时不行），因此这一步必定成功。 2. 记录 τ 的当前状态，记作 τ_0。 3. 对于 σ 中的每个绑定 $x \mapsto t$： (a) 尝试在 τ 中合一 x 和 t。如果成功，则继续步骤 3 的循环。(b) 将 τ 重置为 τ_0，然后再次尝试合一 x 和 t。如果成功，则转到步骤 2。(c) 如果失败，则退出循环，并保留 τ 中的 $x \mapsto t$。

经过此过程后，τ应包含 σ中必要的子集。类似的程序也可用于确定不等式为何被证伪。

[4] https://github.com/MichaelRawson/hopcop，提交 a4a0f66[5] 从带注释的猜想子句（–conj）开始，并防止输入子句重排（–nopaths）

6.3 实验

表 1. 为确定 PUZ005-1 上不存在特定深度的封闭表板而尝试的扩展步骤数。深度为 8 时存在证明。

深度	1	2	3	4	5	6	7
meanCoP	1	4	24	108	535	9,963	6,445,008
hopCoP	1	4	89	495	2,309	10,066	48,517

在理论上费了一番功夫来减少回溯之后，我们希望知道这在实践中是否也有帮助。我们首先手动检查了 meanCoP 和 hopCoP 在取自 TPTP 的 PUZ 领域问题上的行为。meanCoP 会报告穷尽每个迭代加深层级所需成功应用的扩展步骤数量，因此我们也对 hopCoP 进行了检测，使其执行相同操作。表 1 显示了求解 PUZ005-1[6] 所需的步骤数。在较低的迭代加深层级，由于搜索决策和 meanCoP 的引理规则存在差异，结果各有胜负；但在较高层级，hopCoP 通常会明显领先。meanCoP 的推理速率要高得多：我们的实现并未进行高度优化，但我们怀疑，即便经过优化，维护学习到的约束所带来的开销，相较于 meanCoP 仍会显著降低每秒推理次数。

hopCoP 还在几个广受欢迎的一阶基准测试集上与 meanCoP 进行了正面对比：来自 TPTP 9.0 版的 FOF 和合取范式问题0 [39], 、 [1] 中的 MPTP 挑战问题（分别采用 *bushy* 和 *chainy* 变体），以及 *Miz40* ATP-最小化集合 [20], , 其中 *M2k* 是其子集。两个系统对每个问题均设定了 10 秒的时间限制，并将 meanCoP 配置为 --conj --nopaths（以便更好地匹配 hopCoP，见上文）。我们还使用附加的 --cut 参数运行了 meanCoP，并将其称为 !meanCoP：这样可以显著减少回溯，但代价是使 meanCoP 变得不完备。

我们不想据此对两个系统的相对实力做出任何结论，只能说这些数据与以下假设相一致：在每秒推理速度方面，通过减少回溯所获得的收益超过了其开销。表 2 展示了解决问题的数量。读者可能还会对 Färber 关于各种回溯方案的讨论中极为详尽的实验数据感兴趣 [14]。

7 相关工作

与本文最直接相关的工作，是对连接表中回溯施加的各种固定限制 [27,14]：这类方法本质上是不完备的，但很有效。较早的技术，如 失败缓存 [3] ，也能减少回溯，且

[6] PUZ 领域中第一个难度适中的合取范式问题

表 2. hopCoP、meanCoP 和 !meanCoP 在各种基准测试集上于 10 秒内证明的定理。

	M2k	Miz40	bushy	chainy	TPTP
meanCoP	795	7,592	480	157	3,578
!meanCoP	878	9,748	562	337	3,283
hopCoP	1,050	13,040	589	203	4,026

仍然保持完备性，但采用了不同的机制。Goéland 表板系统会在并发分支探索之间交换替换信息 [9]：这可能会减少回溯，但同样采用了不同的机制。将连接驱动搜索编码为 SAT 或 SMT 的方法，例如 ChewTPTP [12]„ 是完备的，也会在证明搜索过程中学习约束，但其行为截然不同，而且大多会预先完成编码。

我们自己也曾尝试将连接方法编码为 SAT 和 SMT [13]，通过用户传播 [7,15]实现。我们发现，即使采用惰性的用户传播器编码，SAT/SMT 求解器也不适合这种证明搜索，因为其内部启发式方法并不了解语义树的当前状态。例如，求解器会非常轻易地决定或传播那些编码了与当前状态完全不相连的某个子表的变量。第 5 节所述的精细编码等方法提升了编码的性能，但允许求解器部分应用推理：我们尚不确定这在性能方面是否有益，但这会造成极大的困惑。

第一作者还研究过在搜索过程中从连接表中发现的子句实例生成SAT 子句 [34], ，作为一种基于实例的方法 [16]。当 SAT 子句集变得不可满足时，这表明输入子句集同样不可满足。虽然这种实例生成方法的扩展可用于影响连接表搜索，但它并非该方法的核心，这与此处的约束学习方法不同。二者在很大程度上是正交的，并且可以有效地结合起来。

另一项高度相关但可能令人困惑的工作是模态及其他语义树中的回跳概念 [17]：该概念及其起源似乎相似，但它用于在寻找模型时避免分支上的逻辑冲突，而不是在寻找封闭表板时避免陷入困境。MeTTeL^2 表板证明器生成器 [40] 对类似的回跳提供通用支持，并将其称为冲突驱动回跳。

8 展望

我们已将一种约束学习方法集成到原型一阶连接定理证明器 hopCoP 中，用于引导搜索并减少回溯。我们观察到，这显著缩小了搜索空间，从而提升了实际性能。代价是内存使用：约束必须存储在某处。根据我们的经验，这种开销并不大，但可能会导致无法在你的 iPod® [30] 上运行 hopCoP。

其他非合流表 tableau 演算系统很可能也能从这种方法中受益。我们也对这类学习与机器学习的交集感兴趣：约束学习可以减少学习得到的启发式方法可选的方案，而良好的学习得到的启发式方法或许能迅速学会有用的约束。

该技术仍有一些可进一步改进之处。最令人困扰的是约束语言中存在的显式位置，这限制了学习到的约束的应用。要消除这一点，就需要在结构等价位置的模意义下检测冲突，我们怀疑这可能难以高效完成。未来的*Lean* 实现或许有助于解决这一问题，或许可以基于最近的认识，即回跳即异常处理 [35]。

致谢。 我们特别感谢 Michael Färber 的 meanCoP 工具，以及他就本主题及相关主题进行的富有启发性的讨论。本研究的全部或部分资金来自 ERC Consolidator Grant ARTIST 101002685、ERC Proof of Concept Grant LEARN 101213411、TU Wien Doctoral College SecInt、FWF SpyCoDe Grant 10.55776/F85、WWTF grant ForSmart 10.47379/ICT22007 以及 Amazon Research Award 2023 QuAT。

利益披露。作者声明，与本文内容相关的利益不存在竞争关系。

参考文献

1. Alama, J., Heskes, T., Kühlwein, D., Tsivtsivadze, E., Urban, J.：通过语料库分析和核方法为数学选择前提。自动推理杂志 **52**(2), 191 5 2 213 2014 10 1007 10817 013 9286 – ( )。 https://doi.org/ . /S - - - . Andrews, P.B.：通过一般配对证明定理。ACM学报 **28**(2), 193–214 (1981 322249 3 10 1145 322248)。https://doi.org/ . / . . Astrachan, O.L., Stickel, M.E.：模型消去定理证明器中的缓存和引理化。载于：Kapur, D.（编）CADE。《计算机科学讲义》，第607卷，第224–238页。施普林格 (1992 10 1007 3 540 55602 8_168)。 https://doi.org/ . / - - - . Backeman, P., Rümmer, P.：有界刚性 e-合一的定理证明。载于：自动演绎——CADE-25——第25届自动演绎国际会议，德国柏林，2015年8月1日至7日，9195，论文集。《计算机科学讲义》，第-卷，第572–587页。施普林格（2015）。 https://doi.org/10.1007/978-3-319-21401-6_39，https://doi.org/ 10 1007 978 3 319 21401 6_39 。 / - - - - . Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.：理论模可满足性。载于：Biere, A., Heule, M., van Maaren, H., Walsh, T.（编）《可满足性手册》第二版，《人工智能与应用前沿》，第336卷，201017 6 1267 1329 2021 10 3233 2，第–页。IOS出版社（）。https://doi.org/ . /FAIA . Bibel, W.：自动定理证明，第nd版。人工智能，Vieweg（1987 16641802 7），https://www.worldcat.org/oclc/ 。Bjørner, N.S., Eisenhofer, C., Kovács, L.：Z3中的自定义理论模可满足性。载于：Dragoi, C., Emmi, M., Wang, J.（编）VMCAI。《计算机科学讲义》，第13881卷，第91–105页。施普林格（2023）。https://doi.org/10.1007/978-3-031-24950-1_5

8. Brand, D.：用修改法证明定理。SIAM计算杂志 **4**(4), 412–430 (1975)。https://doi.org/10.1137/0204036 9. Cailler, J., Rosain, J., Delahaye, D., Robillard, S., Bouziane, H.：Goéland：一种并发的基于语义树的定理证明器（系统描述）。载于：Blanchette, J., Kovács, L., Pattinson, D.（编）IJCAR。《计算机科学讲义》，第13385卷，第359–368页。施普林格（2022）。https://doi.org/10.1007/978-3-031-10769-6_22。 10. D'Agostino, M., Gabbay, D.M., Hähnle, R., Posegga, J.,（编）：语义树方法手册。逻辑、语言与信息杂志 **10**(4), 518–523 (2001)。https://doi.org/10.1023/A:1017520120752 11. Dechter, R.：在约束满足问题搜索过程中学习。载于：第5届全国人工智能会议论文集。第178–185页。Morgan Kaufmann（1986），http://www.aaai.org/Library/AAAI/1986/aaai86-029.php 12. Deshane, T., Hu, W., Jablonski, P., Lin, H., Lynch, C., McGregor, R.E.：在SAT中编码一阶证明。载于：CADE。《计算机科学讲义》，第4603卷，第476–491页。施普林格（2007）。https://doi.org/10.1007/978-3-540-73595-3_35 13. Eisenhofer, C., Rawson, M., Kovács, L.：通过可满足性求解寻找连接。载于：TABLEAUX。《计算机科学讲义》，第15980卷，第82–102页（2025）。https://doi.org/10.1007/978-3-032-06085-3_5。 14. Färber, M.：一种出奇有效的连接表回溯策略。载于：Otten, J., Bibel, W.（编） AReCCa。CEUR研讨会论文集，第3613卷，第23–40页。CEUR-WS.org（2023），https://ceur-ws.org/Vol-3613/AReCCa2023_paper2.pdf 15. Fazekas, K., Niemetz, A., Preiner, M., Kirchweger, M., Szeider, S., Biere, A.：IPASIR-UP：用于冲突驱动子句学习的用户传播器。载于：Mahajan, M., Slivovsky, F.（编）SAT。信息学出版系列，第271卷，第8:1–8:13页。Schloss Dagstuhl - Leibniz-Zentrum für Informatik（2023）。https://doi.org/10.4230/LIPICS.SAT.2023.8 16. Ganzinger, H., Korovin, K.：基于实例化的定理证明新方向。载于：计算机科学逻辑研讨会。第55–64页。IEEE计算机学会（2003）。https://doi.org/10.1109/LICS.2003.1210045 17. Hustadt, U., Schmidt, R.A.：模态语义树中的简化与回跳。载于：de Swart, H.C.M.（编）TABLEAUX。《计算机科学讲义》，第1397卷，第187–201页。施普林格（1998）。https://doi.org/10.1007/3-540-69778-0_2218. ISO/IEC：信息技术——程序设计语言——Prolog——第1部分：通用核心。标准，国际标准化组织（1995）19. Kaliszyk, C.：高效的底层连接表。载于：de Nivelle, H.（编）TABLEAUX。《计算机科学讲义》，第9323卷，第102–111页。施普林格（2015）。https://doi.org/10.1007/978-3-319-24312-2_820. Kaliszyk, C., Urban, J., Michalewski, H., Olsák, M.：定理证明的强化学习。载于：神经信息处理系统大会。第8836–8847页（2018）21. Letz, R., Stenz, G.：模型消去与连接表程序。载于《自动推理手册》（共2卷），第2015–2114页。爱思唯尔和麻省理工学院出版社（2001）。https://doi.org/10.1016/B978-044450813-3/50030-8 22. Loveland, D.W.：通过模型消去进行机械定理证明。ACM学报 **15**(2), 236–251 (1968)。https://doi.org/10.1145/321450.321456 23. Meng, J., Paulson, L.C.：将高阶子句转换为一阶子句。自动推理杂志 **40**(1), 35–60 (2008)。https://doi.org/10.1007/S10817-007-9085-Y 24. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.：Chaff：构建高效SAT求解器。载于：设计自动化会议。第530–535页。美国计算机协会（2001）。https://doi.org/10.1145/378239.379017

25. Nieuwenhuis, R., Rubio, A.: 基于副调制的定理证明。载于：自动推理手册（共2卷），第371–443页（2001）。https://doi.org/10 6 26 1016 978 044450813 3 50009 2 0 1 2 . /B - - / - . Otten, J.: leanCoP . 和 ileancop .：在经典逻辑和直觉主义逻辑中进行高性能精简定理证明（系统描述）。载于：IJCAR。LNCS，第5195卷，第 283 291 2008 10 1007 978 3 540 71070 7_23 –页。施普林格（）。https://doi.org/ . / - - - - . Otten, J.: 限制连接演算中的回溯。AI Commun. **23** (2-3), 159 0464 28 182 2010 10 3233 2010 –（）。https://doi.org/ . /AIC- - . Otten, J.: MleanCoP：一阶模态逻辑的连接证明器。载于：Demri, S., Kapur, D., Weidenbach, C. (编) IJCAR。LNCS，第 8562 卷，第 269–276 页。施普林格（ 2014 10 1007 978 3 319 08587 6_20 ）。https://doi.org/ . / - - - - . Otten, J.: nanoCoP：非子句连接证明器。载于：IJCAR。LNCS，第9706卷，第 302 312 2016 10 1007 978 3 319 40229 1_21 –页。施普林格（）。https://doi.org/ . / - - - - . Otten, J.: 袖珍推理器——在小型设备上进行自动推理。载于：NIK。Bibsys Open Journal Systems，挪威（2018），https://ojs.bibsys.no/index.php/NIK/article/view/512 31 20 . Otten, J.: leanCoP 研究者年——证明器概述。载于：AReCCa。CEUR研讨会论文集，第3613卷，第4–22页。CEUR-WS.org（2023），https://ceur-ws.org/Vol-3613/AReCCa2023_paper1.pdf 32. Paskevich, A.: 采用惰性副调制的连接表。自动推理杂志 **40**(2-3), 179–194（2008）。https://doi.org/10.1007/S10817-007-9089-7, https://doi.org/10 7 33 1007 10817 007 9089 . / s - - - . Prusak, G., Kaliszyk, C.: 实践中的惰性副调制。载于：与第11届自动推理国际联合会议（FLoC/IJCAR 2022）同期举办的自动推理实践方面研讨会论文集，以色列海法，2022年8月11日至12日。CEUR研讨会论文集，第3201卷。CEUR-WS.org（2022），https://ceur-ws.org/Vol-3201/paper3.pdf 34. Rawson, M., Reger, G.: 在模型消去过程中消去模型。载于：Das, A., Negri, S. (编) TABLEAUX。LNCS，第12842卷，第250–265页。施普林格（2021）。https://doi.org/ 10 1007 978 3 030 86059 2_15 . / - - - - . Robbins, E., King, A., Howe, J.M.: 回跳就是异常处理。Theory Pract. Log. Program. **21**(2), 125–144（2021）。https://doi.org/10 1471068420000435 36 1017. /S . Rømming, F., Otten, J., Holden, S.B.: Connections：用于经典、直觉主义和模态连接演算的马尔可夫决策过程。载于：AReCCa。CEUR研讨会论文集，第3613卷，第107–118页。CEUR-WS.org（2023），https://ceur-ws.org/Vol-3613/AReCCa2023_paper8.pdf 37. Silva, J.P.M., Sakallah, K.A.: GRASP——一种新的可满足性搜索算法。载于：ICCAD。第220–227页。IEEE计算机学会 / 美国计算机协会（1996）。https://doi.org/10 569607 38 1109 1996 . / ICCAD. . . Stickel, M.E.: Prolog技术定理证明器。载于：第9届自动演绎国际会议论文集，阿贡，伊利诺伊州，美国，1988年5月23日至26日。LNCS，第310卷，第752–753页。施普林格（1988）。https://doi.org/10 0012881 39 1007. /BFB . Sutcliffe, G.: TPTP问题库及相关基础设施——从合取范式到TH0，TPTP v6.4.0。自动推理杂志 **59**(4), 483–502（2017）。https://doi.org/10 7 40 1007 10817 017 9407 . /S - - - . Tishkovsky, D., Schmidt, R.A., Khodadadi, M.: 语义树证明器生成器MetTeL^2。载于：del Cerro, L.F., Herzig, A., Mengin, J. (编) JELIA。LNCS，第7519卷，第492–495页。施普林格（2012）。https://doi.org/10.1007/978-3-642-33353-8_41

41. Wernhard, C., Bibel, W.: 对证明结构的研究。自动推理杂志 **68**8 42 4 24 2024 10 1007 10817 024 09711（）， （）。https://doi.org/ . /S - - - . Zombori, Z., Urban, J., Brown, C.E.: Prolog技术强化学习证明器（系统描述）。载于：IJCAR。LNCS，第12167卷，第489–507页。施普林格（2020）。https://doi.org/10.1007/978-3-030-51054-1_33