大规模概率推理：触发图来救场

Efthymia Tsamoura 三星
人工智能研究院 英国剑桥
efi.tsamoura@samsung.com

Jaehun Lee 三星 韩国首尔
jaehun20.lee@samsung.com

Jacopo Urbani 阿姆斯特丹自由大学 荷兰阿姆斯特丹 jacopo@cs.vu.nl

arXiv:2304.05459v1 [cs.DB] 11 Apr 2023

摘要

不确定性在数据管理中的作用比以往任何时候都更加突出，尤其是由于机器学习驱动的应用日益重要，而这些应用会生成大量不确定数据库。查询此类数据库的一种广为人知的方法，是将基于规则的推理与不确定性相结合。然而，迄今提出的技术在处理大型数据库时仍然存在困难。本文针对这一问题，提出了一种新的概率推理技术，该技术利用了触发图（TG）——这一概念最近被引入非概率设置。其基本思想是，TG 能够通过避免对谱系进行显式实例化，并将同一事实的相似推导归为一组，从而有效存储概率模型。首先，我们展示如何调整 TG 以支持可能世界语义。随后，我们介绍了高效计算概率模型的技术，并从形式上确立了我们方法的正确性。我们还使用名为 LTG 的原型进行了广泛的实证评估。与其他领先引擎的比较表明，LTG 不仅速度更快，即使与近似推理技术相比也是如此，而且还能对现有引擎无法扩展处理的概率数据库进行推理。

ACM 引用格式：
Efthymia Tsamoura、Jaehun Lee 和 Jacopo Urbani。2023。大规模概率推理：触发图来助阵。载于 *ACM* 会议论文集（*Conference'17*）。ACM，纽约州纽约，美国，20 页。https: //doi.org/10.1145/nnnnnnn.nnnnnnn

1 引言

动机。不确定性是现代数据管理固有的一部分。传统上，不确定性的根源可追溯至从非结构化数据源中挖掘知识 [13, 31, 87, 88] 以及查询传感器测量值 [52], ；但如今，随着神经网络架构的广泛应用，不确定性变得更加普遍。数据库领域对如何高效查询不确定数据进行了广泛研究，诸多开创性成果汇聚于概率数据库（PDBs） [76]。PDBs 使得人们能够在一种优雅的语义框架下查询不确定数据，这种框架称为可能世界语义 [76]。

数据管理领域中另一个拥有悠久历史的概念是基于规则的语言。具体而言，Datalog，这是一种能够以声明式方式表达递归查询的语言，在学术界和工业界都有多种应用 [1, 7, 59]。其中一项应用是查询知识图谱（KGs），这是一种类似图的知识库（KB）。除查询知识图谱外，Datalog 还应用于人工智能和机器学习领域，由此产生了一种称为神经符号人工智能的范式 [56, 79]。例如，Zhu 等人在 [89] 中以及 Huang 等人在 [49] 中使用规则进行视觉问答。

问题。采用基于规则的推理来查询不确定数据，需要以有原则的方式将推理与不确定性相结合。例如，要使用 Datalog 规则查询神经网络 [49],或从 Web 挖掘得到的知识图谱（例如，Google 的知识库[31]）的预测结果，我们需要为 Datalog 语义引入不确定性。尽管在数据库和人工智能领域，将逻辑与不确定性相结合由来已久 [4, 6, 71, 73], ，但当前的概率规则推理方法要么面临可扩展性限制，要么施加了若干语法限制。例如，马尔可夫逻辑网络（MLNs） [71],、概率软逻辑（PSL） [4]和 ICL [65] 要求规则必须是基元的，或满足若干确保非递归性的语法限制。在概率数据库的语境下，虽然已有技术能够在某些情况下高效查询概率数据库，正如 Dalvi 和 Suciu 在 [18], 中所展示的那样，但这些技术仅支持非递归查询。

对于允许在可能世界语义下使用非地面 Datalog 规则的基于规则的语言，情况依然如此 [25, 34, 73]。由于该问题在最坏情况下难以处理，即使对于仅包含数千条事实的概率数据库，推理也可能变得代价高昂、难以承受（讨论见 [2, 89] ）。为应对上述局限，已有研究提出了若干近似技术 [47, 49, 70] 。然而，当需要高置信度时（例如在自动驾驶或医疗保健领域），近似技术不仅不切实际，而且仍无法扩展到一定规模以上。

作为近似技术的替代方案，[78] 的作者最近提出了 ΔTcP，这是一种基于来源半环相关研究 [43] 以及逻辑编程领域广为人知的 TcP 算子 [86]的新技术。ΔTcP 通过将精确概率推理转化为非概率推理 [78]，提升了其可扩展性。尽管在运行时间方面优于以往方法，ΔTcP 仍面临若干性能瓶颈。首先，ΔTcP 在依据规则进行推理后，必须执行布尔公式比较以确保终止（**L1**）。然而，在查询图时，布尔比较的开销可能非常大 [34, 69, 86]。其次，ΔTcP 可能会保留同一公式的多个副本（**L2**），从而增加内存消耗。除这两项限制外，ΔTcP 还引入了另一项限制：将规则改写为更复杂的规则，并维护额外的数据结构（**L3**）。这些限制在某些情况下会造成性能瓶颈。例如，在我们的实验中

特此授权：在不收取费用的情况下，为个人或课堂使用而制作本作品全部或部分内容的数字或纸质副本，但前提是不得为营利或商业利益而制作或分发副本，并且副本首页须载有本声明及完整引文。必须尊重本作品中由非 ACM机构拥有版权的组件。经注明出处，允许摘编。若要以其他方式复制、再版、发布到服务器或重新分发到列表，则须事先获得具体许可和/或支付费用。请通过 permissions@acm.org 申请许可。*Conference'17*，*2017*年7月，美国华盛顿特区 © 2023 计算机协会。ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...15.00 美元 https://doi.org/10.1145/nnnnnnn.nnnnnnn

我们使用 ΔTcP 在广为人知的基准测试 LUBM [46], 上测量发现，**L1** 和**L3** 带来的开销最高可占总运行时间的 96%。

我们的方法。 本文介绍了一种基于概率规则进行推理的新技术，该推理采用可能世界语义，并克服了上述三个局限。为此，我们展示了如何以比当前可行方式更具可扩展性的方式对概率数据库进行推理。

我们的方法基于触发图 （TGs）。触发图是一种最近为非概率规则推理引入的结构 [77]。TGs 是一种有向无环图，捕获了使用一组规则计算非概率数据库模型时应执行的所有操作，即计算数据库的一个扩展，使所有规则在逻辑上都得到满足。已有研究表明 [77] ，借助避免冗余推导的能力，使用 TGs 进行推理比采用先前技术进行推理高效得多 [10, 29] 。

TGs 的一个局限是，它们不能直接用于概率推理。在本文中，我们展示了通过适当的修改，TGs 可以维护推导的来源信息，而这些信息可直接用于计算其概率 [53]。通过这样做，概率推理可以以一种克服上述局限 **L1–L3** 的方式实现。关于**L1**，TGs 消除了执行布尔公式比较的要求。关于**L2**，将推导来源信息存储在 TG 中，无需多次存储同一公式。此外，我们展示了如何将多个推导树合并为一棵，以节省空间并提高运行时间。最后，关于**L3**，TGs 允许我们原生维护来源信息，从而无需将规则重写为更复杂的形式，也无需维护额外的结构。

我们将该技术实现为一个名为谱系触发图（LTG）的新引擎，并将其性能与领先的引擎进行比较，即 ProbLog2 [74, 82] 和 vProbLog [78]。我们的实证评估考察了来自 Web（LUBM [46], DBpedia [12]和 Claros [66]）以及概率逻辑编程社区（ Smokers [30]）的场景。此外，我们还使用热门的真实世界知识图谱（YAGO 和 WN18RR [26]）以及通过最先进技术挖掘的规则（AnyBurl [57]）开展了实验。最后，我们还考察了一个名为 VQAR [49]的近期基准测试。在 VQAR 中，概率事实由神经网络推导得出，而规则则用于回答有关图像 [49]的查询。由于推理会导致推导数量激增，这项基准测试对先前的研究成果构成了挑战。事实上，这项基准测试促成了 Scallop [49],的诞生，它是一种近期提出的近似概率推理引擎，性能达到最先进水平。我们将 LTG 与 Scallop 进行了比较，发现尽管 Scallop 不会搜索所有解释，LTG 的表现仍常常优于 Scallop。值得注意的是，LTG 是唯一能够计算 VQAR 完整概率模型的引擎，这得益于它维持紧凑模型表示的能力。

总的来说，我们的实验结果表明，我们的方法在运行时间和 RAM 消耗方面都优于其他引擎，而且往往优势显著。此外，在多个场景中，LTG 可能决定了能否使用规则回答概率数据库上的查询。

总而言之，我们的贡献如下：

- 我们提出了一种基于分布语义和触发图（TG）的新技术，用于对大型概率数据库进行推理。
- 我们提出了一种扩展，通过压缩避免推导数量的组合爆炸。
- 我们证明了该方法的正确性，并表明它能够像已有方法（[78, 86]）一样提供随时可用的界。
- 我们在一个名为 LTG 的新引擎中实现了该技术，并使用来自不同领域的一组基准测试，将其性能与最先进的引擎进行了比较。

2 预备知识

我们首先简要回顾一些与逻辑和（概率）基于规则的推理相关的基本概念。

一个术语 要么是常量，要么是变量。原子 的形式为$p(t_1, \ldots, t_n)$，其中 p是一个 n元谓词，且每个 t_i都是一个术语。如果一个原子中的所有术语都是常量，则称该原子是基的。基原子也称为事实。术语映射 σ是从术语到术语的（可能是部分的）映射；我们用 $\sigma = \{t_1 \mapsto s_1, \ldots, t_n \mapsto s_n\}$表示，对于 $1 \leq i \leq n$，$\sigma(t_i) = s_i$。设 α是一个术语、一个公式，或一个术语集合或公式集合。那么，通过将 α中每个同时出现在 σ定义域中的术语 t替换为 $\sigma(t)$，即可得到 $\sigma(\alpha)$（即，定义域之外的术语 σ保持不变）。我们将 $\sigma(\alpha)$称为 α的实例化。符号 $\models$表示逻辑蕴涵。对于一组基原子 I和一个原子 α，当 $\alpha \in I$时， $I \models \alpha$成立。符号$\equiv$表示逻辑等价。

A Datalog规则 是形如以下形式的全称量化蕴含式。

$$p(\mathrm{X}) \leftarrow \bigwedge_{j=1}^{n} p_j(\mathrm{X}_j). \tag{1}$$

上述内容中，X 和 Xj, $1 \leq j \leq n$ 是变量向量，X 中的每个变量也出现在某个 Xj 中。此后，我们将 Datalog 规则称为规则。我们将规则的右侧称为其前提，左侧称为其结论。

逻辑程序。 一个（非概率）逻辑程序 P 是一对 (R，F)，其中 R 是规则的集合，F 是事实的集合。程序 P 的赫布兰基底 HB(P) 表示使用 P 中出现的所有常量和谓词所能计算出的所有基原子的集合。P 的解释 是将 P 的赫布兰基底中的每个原子赋值为真或假的一种赋值。等价地，我们可以将解释视为 HB(P) 的一个子集，其中只包含被赋值为真的原子。如果对 P 中的每条规则 r， $I \models r$都成立，那么该解释就是 P 的一个模型 。P 的最小 赫布兰德模型是 P 的所有模型中原子数量最少的模型。每个由 Datalog 规则组成的程序都存在有限模型。我们用 P $\models \alpha$或 (R，F) $\models \alpha$表示 R $\cup$F $\models \alpha$，其中 α是一个基原子。

查询使用一个新的谓词Q 来定义。常量元组 a 是相对于程序 (R，F) 而言的 Q 的答案，当且仅当 R $\cup$F $\models$Q(a)。上述定义使我们能够表示合取查询（CQs）[14] ，方法是在结论中引入一条定义 Q 的规则 [11]。

概率数据库。 一个元组独立概率数据库 （PDB）D 是一对（F π），其中 F 是一组事实。每个事实都被视为一个相互独立的伯努利随机变量，以 $\pi(f)$ 的概率变为真（相应地，以 $1 - \pi(f)$ 的概率变为假） [76]。下文中，我们将写作 $\pi(f) :: f$，表示事实 f及其为真的概率。一个概率数据库会在所有数据库实例上诱导出一个分布，我们称之为可能世界。F 的每个子集都是一个可能世界。将

数据库事实视为相互独立的随机变量后，我们可以将 D 中可能世界 C 的概率 Pr(C) 计算为：C 中为真的事实的概率之积，乘以 C 中为假的事实的概率之积。于是，D 中公式 φ 的概率就是所有满足 φ 的可能世界的概率之和。

概率逻辑程序[85] 通过基于规则的推理扩展概率数据库。概率逻辑程序，简称概率程序，是一个三元组 P =(R、F π)，其中 R、F 和 π的定义同上。P 中公式 φ的概率定义方式与概率数据库类似。不过，这一次我们考虑 (F π) 的所有可能世界，这些可能世界与规则 R 一起蕴含 φ:

$$\sum_{C\subseteq\mathcal{F}|C\cup\mathcal{R}\models\varphi} \Pr(C) \tag{2}$$

请注意，也可以通过向规则前提添加额外的"虚拟"事实，并将其概率设为与规则相等，从而为规则分配概率 [24]。

P 中地面原子 α的一个解释是 F 的一个极小子集 C，使得它与 R 一起蕴含 α。我们使用构成该解释的原子的合取来表示解释。P 中 α的谱系是其在 P 中所有解释的析取。

示例1. 考虑描述图可达性的规则集 $\mathcal{R}$

$$p(X,Y) \leftarrow e(X,Y) \tag{r_1}$$
$$p(X,Y) \leftarrow p(X,Z) \wedge p(Z,Y) \tag{r_2}$$

根据 $\mathcal{R}$，如果存在从 X到 Y的一条边，或者存在从 X到 Z和从 Z到 Y 的路径，则存在一条从 X到 Y的路径。再考虑集合 $\mathcal{F}$，其中包含概率事实 $e(a,b)$、 $e(b,c)$、 $e(a,c)$和 $e(c,b)$。每个事实 f为真的概率记为 $\pi(f)$。

考虑事实 $p(a,b)$。它在 P =$(R, F\ \pi)$ 中的概率等于所有包含事实 $e(a,b)$，或包含事实 $e(a,c)$ 和 $e(c,b)$ 的可能世界的概率之和。因此， $e(a,b)$ 和 $e(a,c) \wedge e(c,b)$ 是 P 中 $p(a,b)$ 的两种解释，而 $e(a,b) \vee e(a,c) \wedge e(c,b)$ 是其谱系。

在本节余下部分，我们固定一个程序 P =(R, F π)。概率程序有多个最小赫布兰德模型：对于 PDB (F π) 的每个可能世界 C，逻辑程序 (R, C) 的最小赫布兰德模型也是 P 的一个（最小赫布兰德）模型。

概率推理。 最先进的概率推理技术，例如 ProbLog [78, 86] 和 Scallop [49] 提出的方法，建立在关于来源半环 [43]的研究之上。

令 V 表示与 F 中事实相关联的布尔变量集合。其思路是，首先为每个事实 α关联一个关于 V 的布尔公式 λ_α，以表示 α 的来源信息 [43], , 然后通过对 λ_α进行加权模型计数 (WMC) [72] 来计算 Pr(α)。为了计算 α和 λ_α，Vlasselaer 等人借鉴了自底向上 Datalog 求值的思想，并引入最小参数化模型 的概念，以及用于计算该模型的 $Tc_\mathcal{P}$ 算子 [86]。P 的一个最小参数化模型，对于 P 的任一最小赫布兰德模型中出现的每个原子 α，都包含一个形如 (α, λ_α) 的二元组。我们通常将最小参数化模型称为概率模型。

$Tc_\mathcal{P}$ 按轮次进行，其中每一轮 k都会针对每个原子 α计算一个公式 λ_α^k，该公式编码了深度为≤ k的原子 α的所有推导。每一轮包括三个步骤：推导步骤（**DE**），一个

表1： $Tc_\mathcal{P}$**与** $\Delta Tc_\mathcal{P}$**前3轮的公式**。ω**表示公式** $\lambda^2_{p(a,\ b)} \wedge \lambda^2_{p(b,\ b)} \vee \lambda^2_{p(a,\ c)} \wedge \lambda^1_{p(c,\ b)}$。

R	Atom	μ^i	λ^i
1	$p(a,\ b)$	$e(a,\ b)$	$e(a,\ b)$
	$p(b,\ c)$	$e(b,\ c)$	$e(b,\ c)$
	$p(a,c)$	$e(a,c)$	$e(a,\ c)$
	$p(c,b)$	$e(c,b)$	$e(c,b)$
2	$p(a,b)$	$\lambda^1_{p(a,\ c)} \wedge \lambda^1_{p(c,\ b)}$	$e(a,c) \wedge e(c,b) \vee e(a,b)$
	$p(a,c)$	$\lambda^1_{p(a,\ b)} \wedge \lambda^1_{p(b,\ c)}$	$e(a,\ b) \wedge e(b,\ c) \vee e(a,\ c)$
	$p(b,b)$	$\lambda^1_{p(b,\ c)} \wedge \lambda^1_{p(c,\ b)}$	$e(b,c) \wedge e(c,b)$
3	$p(a,b)$	ω	$\mu^3_{p(a,b)} \vee \lambda^2_{p(a,b)} \equiv \lambda^2_{p(a,b)}$
	$p(b,c)$	$\lambda^2_{p(b,\ b)} \wedge \lambda^1_{p(b,\ c)}$	$\mu^3_{p(b,\ c)} \vee \lambda^2_{p(b,\ c)} \equiv \lambda^2_{p(b,\ c)}$
	$p(a,c)$	$\lambda^2_{p(a,\ b)} \wedge \lambda^1_{p(b,\ c)}$	$\mu^3_{p(a,c)} \vee \lambda^2_{p(a,\ c)} \equiv \lambda^2_{p(a,\ c)}$
	$p(b,b)$	$\lambda^2_{p(b,\ b)} \wedge \lambda^2_{p(b,\ b)}$	$\mu^3_{p(b,b)} \vee \lambda^2_{p(b,\ b)} \equiv \lambda^2_{p(b,\ b)}$

聚合步骤 **(AG)**，以及公式更新步骤 **(FU)**。**DE** 使用目前为止推导出的原子实例化R中的规则，并根据每个规则实例化计算一个布尔公式。然后，对于每个原子 α，**AG** 通过对所有 在**DE**处为 α 计算出的公式进行不相交化，计算出一个新公式 μ_α^k。最后，如果 $\mu_\alpha^k \not\equiv \lambda_\alpha^{k-1}$，**FU** 计算一个新公式$\lambda_\alpha^k = \mu_\alpha^k \vee \lambda_\alpha^{k-1}$，否则将其设为 $\lambda_\alpha^k = \lambda_\alpha^{k-1}$。该技术在第 k 轮终止，此时所有在第 k轮计算出的公式都与在第（$k-1$）轮计算出的公式逻辑等价。

示例*2*。我们通过示例*1*演示 Tc_P。第一轮中，Tc_P通过使用F中的事实实例化 r_1，计算所有长度为一的路径。例如，实例化 $p(a,b) \leftarrow e(a,b)$表示存在一条从 a到 b的路径，因为存在一条从 a到 b的边。因此，$\lambda^1_{p(a,b)} = e(a,b)$。第二轮中， Tc_P计算所有长度至多为二的路径。在这一轮中，计算出实例化 $p(a,b) \leftarrow p(a,c) \wedge p(c,b)$。该实例化表示存在一条从 a到 b的路径，因为存在一条从 a到 c的路径，以及一条从 c到 b的路径。由于$\lambda^1_{p(a,c)} = e(a,c)$和$\lambda^1_{p(c,b)} = e(c,b)$，公式 $e(a,c) \wedge e(c,b)$是由该规则实例化和FU集合$\lambda^2_{p(a,b)} = e(a,c) \wedge e(c,b) \vee e(a,b)$计算得出的。随后， Tc_P开始第三轮，计算所有长度至多为三的路径。由于第三轮中计算出的所有公式都与第二轮中计算出的公式逻辑等价， Tc_P终止。表*1*列出了前三轮中计算出的一些公式。为简洁起见，表*1*中的 μ^i 列没有列出所有规则实例化对应的公式，而只列出由至少涉及一个"新鲜"事实的规则实例化所计算出的公式；也就是说，该事实要么是在上一轮推导出的，要么其公式在上一轮中已更新。例如，表*1*没有列出第二轮中事实$p(b,c)$和 $p(c,b)$对应的公式，因为此时这些事实只能通过r_1和数据库事实推导得出。正如我们将在下文讨论的，表*1*中的推导反映了 ΔTc_P的推导。

计算概率模型颇具挑战，因为我们需要为每个原子存储其谱系中的所有解释，而这些解释的数量可能呈指数级增长 [24]。此外，计算给定谱系的概率是#P 难的 [81]。因此，对于某些最坏情况的输入，计算谱系或其概率可能耗时过长，或耗尽内存。尽管我们的方法并未改变

问题的最坏情况复杂度，但其目标与 $\Delta Tc\mathcal{P}$ 类似，是提高可扩展性，从而显著减少最坏情况输入的数量。

$\Delta Tc\mathcal{P}$ 解决了 $Tc\mathcal{P}$ 所面临的问题，即重复计算相同的规则实例化。例如，$p(a,b) \leftarrow e(a,b)$ 在 $Tc\mathcal{P}$ 的第二轮次和第三轮次中都被计算。为避免这些重复计算，[78] 引入了 $\Delta Tc\mathcal{P}$，将其作为 $Tc\mathcal{P}$ 的扩展，灵感来自半朴素求值（SNE）。SNE 是一种广为人知的 Datalog技术，它将第 k 轮次中的规则实例化限制为至少涉及一个原子，且该原子的谱系在第（$k-1$）轮 [1] 中得到更新的实例化。[78] 中的作者还提出了 $\Delta Tc\mathcal{P}$ 的声明式实现：通过重写规则、引入辅助原子，并添加用于填充这些原子的新规则来实现。

3 动机

例2揭示了 $Tc\mathcal{P}$ 和 $\Delta Tc\mathcal{P}$ 的若干局限。首先，$Tc\mathcal{P}$ 和 $\Delta Tc\mathcal{P}$ 都会在每一轮结束时进行布尔公式比较。例如，它们都会在第二轮结束时，将公式 $\lambda^1_{p(\quad\quad\quad

定义 1。 *[从 [77]]* 规则集合$\mathcal{R}$的执行图 （*EG*）是一个无环的、节点和边均带标签的有向图 $G=(V,E,$ 规则 $\ell)$，其中 V 和 E 分别是图的节点集合和边集合，而规则 和 ℓ 是节点和边的标记函数。每个节点 v ***(i)***都标记有某条规则，记作规则(v)，该规则来自 $\mathcal{R}$；并且 ***(ii)*** 只有当规则(v) 的规则体中的第 j个谓词等于规则(u) 的头谓词时，才可以存在一条从节点 u 指向节点 v、形式为 $u \rightarrow_j v$ 的带标签边。

图1b展示了示例1中规则的解释图 G。对于规则集合 R，解释图G描述了在事实集合F 上执行 R 中规则的计划。如果 G包含一个能够计算最小 Herbrand 模型的计划，那么我们称 G为 TG。事实上，图1b中的图就是一个TG。使用 G对 F进行推理涉及以自底向上的方式遍历该图，使用与节点 v 关联的父节点所关联的事实来实例化与节点 r 关联的每条规则，并将结果存储在 v中。如果节点 v 没有父节点，那么我们称其为源节点，并使用 F中的事实来实例化 r。节点的深度 v 在 G 中，是以 v为终点的最长路径中节点的数量。如果G 是空图，则G 的 深度 d(G)为0；否则，它是G中节点的最大深度。

下面，我们展示一个关于执行图（以及触发图）进行推理的示例。

示例 3。 我们使用图*1b*中的解释图，演示如何对示例 *1* 中的事实进行推理。推理从 v_1 开始，接着处理 v_2，最后处理 v_3、 v_4 和 v_5。由于 v_1 没有入边，我们会对 $\mathcal{F}$中的所有事实实例化 r_1 的前提（与规则 v_1 关联的规则）。在实例化与 v_1 关联的 r_1 时，会计算术语映射[1] $h_1=(a,b)$、$h_2=(b,c)$、 $h_3=(a,c)$和 $h_4=(c,b)$。使用每个术语映射 h_i 对 r_1 的结论进行实例化后得到的所有事实，都会存储在 v_1 中。

对 v_1 进行推理后，接下来需要考虑的节点是 v_2，它与规则 r_2 相关联。边 $v_1 \rightarrow_1 v_2$ 和 $v_1 \rightarrow_2 v_2$ 规定， r_2 前提中的两个原子都必须使用存储在 v_1 中的事实进行实例化。项映射[2] $h_5=(a,b,c)$、 $h_6=(b,c,b)$ 和 $h_7=(a,c,b)$ 将在 v_2 和 F 的上下文中实例化 r_2，而派生事实 $p(a,c)$ 、 $p(b,b)$ 和 $p(a,b)$ 将存储在 v_2 中。

4.1 用于概率推理的执行图

EG（或 TG）的结构对应于推断 Herbrand 模型中事实的一系列步骤。通过追溯规则实例化，我们可以计算事实的所有推导树并提取其谱系。接下来，我们不失一般性地假设 EG 是规范的：非叶节点 v 与形式为 $p(\mathbf{X}) \leftarrow \bigwedge_{j=1}^{n} p_j(\mathbf{X}j)$ 的规则相关联，并且对于每个 $1 \le j \le n$，EG 都包含一条形式为 $u_j \rightarrow_j v$ 的边。如 [77]，所示，我们总是可以将规则改写为一种能够生成规范 EG 的形式。

与非概率环境中的推理相比，一个主要区别在于，现在我们不再将一组事实存储在节点中，而必须存储与其关联的推导树。节点中的树取决于特定上下文：解释图中的祖先节点。

定义2。 令 $(\mathcal{R},\mathcal{F})$ 为一个程序，G为$\mathcal{R}$ 的规范解释图，且 v 为 G 中与规则 r 相关联的一个节点。集合$\mathsf{T}(\alpha,v,\mathcal{F})$ 按如下方式构造：

[1]记号 $h_i=(c_1,c_2)$是 $h_i=\{X \mapsto c_1, Y \mapsto c_2\}$的简写。[2]记号 $h_i=(c_1,c_2,c_3)$是 $h_i=\{X \mapsto c_1, Z \mapsto c_2, Y \mapsto c_3\}$的简写。

算法1 PReason（P），其中$\mathcal{P}$ =（R，$\mathcal{F}$ π）

1: $k \coloneqq 0$; G^0 是 R 的空解释图;
2: **做**
3: $k \coloneqq k+1$;
4: 计算 G^k 从 G开始 G^{k-1} 以增量方式
5: **对于每个** 节点 v的深度 k **执行**
6: tset(v，F) $\coloneqq \emptyset$
7: **对于每个** $\mathsf{T}(\alpha,v,\mathcal{F}) \neq \emptyset$对于某些 α**执行**
8: **对于每棵**推导树 $\tau \in \mathsf{T}(\alpha,v,$ F) **执行**
9: **如果** τ相对于 α 不是冗余的，**则**
10: **将** τ添加到tset(v，$\mathcal{F}$)
11: **如果** tset(v，$\mathcal{F}$) $=\emptyset$ **则从** G **中移除** v
12: **当** d(G^k) $\neq$ d(G^{k-1})
13: **返回** $G^k(\mathcal{F})$

- 如果 v是源节点，则对于 r的每个实例化 $\alpha \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n$，使得每个 α_i都在 F 中，$T(\alpha,v,F)$ 包含一棵根为 α且边为 $\alpha_i \rightarrow \alpha$的树；否则，
- 对于 r的每个实例化 $\alpha \leftarrow \alpha_1 \wedge \ldots \wedge \alpha_n$，使得对于每个 $T(\alpha_i,u_i,F) \neq \emptyset$，$T(\alpha,v,F)$ 都包含：对于从 $T(\alpha_1,u_1,F) \times \cdots \times T(\alpha_n,u_n,F)$ 中选取的树 $(\tau_1,\ldots,\tau_n)$ 的每种组合，都有一棵根节点为 α、且从每棵 τ_i的根到 α都有一条边的树（回想一下，G 是规范的）。我们将 $T(\alpha,v,F)$ 中的树称为推导树。

图1a展示了使用图1b中的解释图对示例1中的事实进行推理时计算出的推导树 τ_1–τ_{11}。在全文中，我们用 root(τ) 表示推导树 τ根部的事实，用children(v) 表示其根通过一条边连接到节点 v的子树。此外，我们为每个节点标注一个标签，说明如何从其（可能的）祖先推导出该事实。默认标签是与，表示节点中的事实需要所有祖先事实才能推导出来。在下一节中，我们将介绍另一个标签，即或，用于指定替代推导。

与非概率设置中的推理相比，另一个主要区别在于终止：当当前轮次中推断出的所有事实都冗余时，推理便会终止。在非概率设置中，如果某个事实此前已经推导出，那么该事实就是冗余的。但在概率设置中，这一条件会损害正确性。要判断某个推导是否冗余，我们必须考虑与其关联的推导树。事实证明，只需丢弃推导树 τ，如果 α在 τ中出现不止一次即可。如果符合这一条件，那么我们就称 τ相对于 α是冗余。

现在，我们可以介绍我们的推理过程了，该过程构造适用于概率推理的解释图。该过程称为 概率推理 （PReason），如算法1所示。给定一个概率程序 P =(R，F π) 作为输入，该过程分轮次执行。在每一轮 k中，它首先计算深度为 k 的解释图（第4行）。这一计算以增量方式进行：也就是说，该过程先处理深度为 $<k$ 的所有节点，然后通过对这些节点应用规则的实例化，添加所有可能构造出的节点。接着，算法1执行与深度为 k 的节点相关联的规则。执行与节点 v 相关联的规则包括计算相应的推导树（第7行），并将其中的一个子集存储在集合 tset(v，F) 中，该集合包含与

v 相关联的树（第10行）。如上文所述，是否丢弃一棵树的判断在第9行进行。最后，如果 tset(v, F) 为空，则移除节点（第11行）。当第 k 轮中的所有节点都被移除后，PReason 结束。

示例 4. 我们通过贯穿全文的示例演示算法*1*。第一次迭代中，算法*1*计算出一个仅包含图*1b*中节点 v_1 的解释图，并将树 τ_1 到 τ_4 存储在 tset($v_1, \mathcal{F}$) 中。第二次迭代中，算法*1*计算出一个包含图*1b*中节点 v_1 和 v_2 的解释图，并将树 τ_5、τ_6 和 τ_7 存储在 tset($v_2, \mathcal{F}$) 中。第三次迭代中，算法 *1* 将节点 v_3、v_4 和 v_5 添加到上一轮计算出的图中，见图 *1b*。我们重点来看 v_3。尽管在 v_3 和 $\mathcal{F}$ 的上下文中存在一棵推导树 τ_8，tset(v_3, $\mathcal{F}$) 仍为空。这是因为 τ_8 根节点中的事实也出现在一个内部节点中。出于类似原因，没有推导树被添加到 tset($v_4, \mathcal{F}$) 和 tset($v_5, \mathcal{F}$) 中，因此算法*1*终止。

请注意，由于推导组织在 TG 内部，我们不需要完整存储 tset(v, F) 中的所有树。相反，我们可以利用结构共享 [80]，仅存储推导树的根以及指向其祖先的指针。为了计算谱系，我们可以通过遍历 TG 动态重建推导树。

4.2 正确性

PReason(P) 的正确性通过一系列步骤得以证明。首先，我们展示如何使用推导树来计算原子的谱系。本质上，在解释图上进行推理时生成的推导树使我们能够重构类似于 [78] 中模型的模型。

在下方的引理 1 中，我们考虑算法1的一种简化版本：忽略第 9 行步骤中的条件，以便将在第 8 行访问的每棵树都添加到节点 v_0。稍后我们将修正这一假设。目前，在这一简化条件下，我们用 G^i(F) 表示使用算法1对 F 和 P 进行推理时，存储在 G 中深度为 i 的节点内的推导树。对于一棵推导树 τ，我们还用 $\phi(\tau)$ 表示对 τ 中的叶节点取合取后得到的布尔公式。类似地，我们考虑 $Tc_{\mathcal{P}}$ 的一种简化版本，该版本避免在每轮结束时执行布尔公式检查，并将第 i 次迭代结束时计算出的实例记为 I^i，其中 $\mathrm{I}^0 = \{(f, f) \mid p :: f \in \mathrm{F}\}$。事实证明，这两个简化算法所计算的谱系公式之间存在一一对应关系。

引理 1. 对于每个 $i \geq 0$，$(\alpha, \lambda^i_\alpha) \in \mathcal{I}^i$ 当且仅当 $\bigvee_{j=1}^{m} \phi(\tau_j) \equiv \lambda^i_\alpha$，其中 τ_1、……、τ_m 都是 $G^i(\mathcal{F})$ 中以事实 α 为根的树。

引理 1 表明，要计算原子α的概率，只需收集存储在G(F) 中的 α 的所有推导树，计算每棵树得出的公式，最后取这些公式的析取。

现在，我们来讨论终止。由引理 1 可知，判断何时终止可归结为判断：对于原子 α，推导树 τ 的公式$\phi(\tau)$ 是否因另一棵针对 α 的推导树 τ' 的公式 $\phi(\tau')$ 而在逻辑上冗余，即 $\phi(\tau) \vee \phi(\tau') \equiv \phi(\tau')$ 是否成立。显然，当 τ 以 τ' 为子树时，$\phi(\tau')$ 是公式 $\phi(\tau)$ 中的一个合取项，因此 $\phi(\tau) \vee \phi(\tau') \equiv \phi(\tau')$。当上述情况成立时，我们称在 τ 下对 α 的推导相对于在 τ' 下对 α 的推导是多余的 。

命题1。 对于 α、τ与 τ'的两棵推导树，如果 τ'是 τ的子树，则 $\phi(\tau) \vee \phi(\tau') \equiv \phi(\tau')$成立。

要检测α的多余推导，只需检查α 是否出现在其新计算的推导树τ的内部节点中，即检查 τ 相对于 α是否是冗余的，正如我们在上一节中形式化的那样。因此，重新启用第 9行中的检查是安全的（我们在讨论开始时禁用了该检查），因为其任务正是丢弃冗余推导。这样，当深度为 k 的所有节点都为空时，推理就会终止。

引理 2. PReason($\mathcal{P}$) 对每个具有有限赫布兰基底的概率程序 $\mathcal{P}$ 均会终止。

对于原子 α，我们将其在 G(F)中的谱系定义为：对 α 在 G(F) 中的推导树所对应的公式进行析取后得到的公式。现在，我们已经可以介绍谱系触发图的概念。

定义 3. 对于概率程序 $\mathcal{P} = (\mathcal{R}, \mathcal{F}, \pi)$，若一个解释图$G$是 $\mathcal{P}$ 的谱系 TG，则称其为 $\mathcal{P}$的解释图；如果对于每个原子 $\alpha \in \mathrm{HB}(\mathcal{P}) \setminus \mathcal{F}$，$\alpha$在 $G(\mathcal{F})$ 中的谱系与 α在 $\mathcal{P}$中的谱系逻辑等价，则满足此条件。

以下结果确立了算法1的正确性，这可由引理 1、命题1和引理 2 得出。

定理 2. PReason($\mathcal{P}$) 是任意概率程序 $\mathcal{P}$ 的谱系 TG。

此外，在对 P 进行推理的每一轮 k 中，根据原子 α在 G^k(F) 中的谱系计算出的概率，是 α实际概率的下界。

推论3。对于每个概率程序$\mathcal{P}$ 以及每个原子$\alpha \in HB(\mathcal{P})$，其在 $G^k(\mathcal{F})$ 中的谱系概率小于$\mathcal{P}$ 中 α的概率。

该推论直接由上文的引理 1 以及谱系的单调性得出。

5 压缩谱系

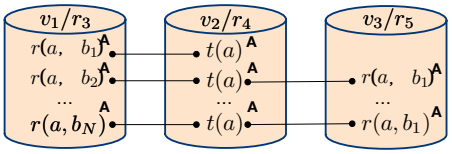
算法1的一个局限是，一个节点可能包含同一事实的多个推导树。上述情况可能导致推导数量呈指数级增长，因为这些推导树中的每一个都可能在未来的规则实例化中被考虑。这种现象在实践中可以观察到。例如，在等式规则（*sameAs* [11, 58]）下进行推理时就可以观察到这一现象。在 VQAR 中也能观察到，详见第 6 节。

为解决这一问题，我们提出了一种优化方法，将此类树合并为一棵树，以减少内存消耗和运行时间。我们提供了一个示例进行说明。

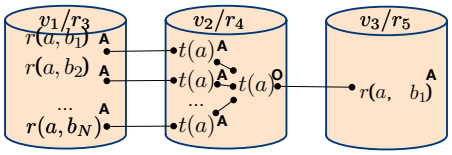
示例 5。 考虑一个包含以下三条规则的程序

$$r(X, Y) \leftarrow q(X, Y) \qquad (r_3)$$
$$t(X) \leftarrow r(X, Y) \qquad (r_4)$$
$$r(X, Y) \leftarrow t(X) \wedge s(X, Y) \qquad (r_5)$$

并令$\mathcal{F}$为一个包含 $q(a, b_i)$（其中 $1 \leq i \leq N$）以及 $s(a, b_1)$的事实集合。图 *2a* 展示了相应的谱系 TG Γ



(a) 所有推导树均单独存储。



(b) 所有以 $t(\alpha)$ 为根的推导树都已折叠。

图 2：在示例 5 中存储 Γ 的不同方式。上标 A（O）表示该事实标记为 与 （或）。为清晰起见，未显示指向 $\mathcal{F}$ 中事实的边。

程序以及存储在 Γ 中节点内的推导树（为清晰起见，指向 $\mathcal{F}$ 中事实的边未显示）。节点 v_2 存储了 N棵不同的树，其根为 $t(a)$。$N-1$棵此类树与 $s(a, b_1)$结合后，根据规则 r_5生成根为 $r(a, b_1)$的 $N-1$棵树，并存储在 v_3 中。

示例 5 表明，更节省空间的技术是在节点 v_2 中只保留 $t(a)$ 的推导树，这样只需将一棵根为 $r(a, b_1)$ 的树添加到 v_3。这样做的挑战在于要记住，$t(a)$ 可以通过 N 种不同的方式推断出来。

为了解决这个问题，我们首先引入一个额外的标签，称为 OR。回想一下，在第 4 节中，推导树中的所有节点都标记为 AND。直观来说，标记为 OR 的事实与它们不同，因为只要有一个祖先成立，它们就成立。接下来，我们按如下方式定义将多个推导树折叠为一棵树的过程。

定义 4. 设 $\tau_1, \tau_2, \ldots, \tau_m$ 是一组根事实相同、且根事实为 α 的推导树。那么，折叠$(\tau_1, \tau_2, \ldots, \tau_m)$，其中 $m > 1$，是按如下方式定义的推导树 τ：

(1) 根(τ) 是事实 α，并且标记为 或；以及 *(2)* 从 根(τ) 到每个 根(τ_i) 都存在一条边。

通过折叠推导树，我们可以减少后续规则实例化的次数。例如，图 2b 展示了这一操作对示例 5 的影响。在这里，v_2 中所有以 $t(a)$ 为根的树都被折叠为一个条目。当规则 r_5 应用于 v_2 时，$t(a)$ 只会被考虑一次，因此推导出一棵树，而不是 $N-1$ 棵树。

如果不存在标记为 OR 的节点，那么收集推导树的谱系只需收集其叶节点。否则，必须修改这一过程，将标记为 OR 的节点引入的分支展开为多棵树。我们将推导树展开的概念形式化如下。

定义 5. 推导树 $\tau(\tau)$ τ（其子节点(根(τ)) $=\{\tau_1, \ldots, \tau_m\}$）的展开展开为：

★ $\{\tau\}$，如果 τ 中没有节点标记为 或；或者

† $\bigcup_{\tau_i}$展开(τ_i)，如果 根(τ)的标签为 或；或者，‡ Δ，其中，对于来自 展开(τ_1) $\times \cdots \times$展开(τ_m)的每个树组合 $(\delta_1$、…、$\delta_m)$，Δ 包含一棵推导树

Algorithm 2 PCOR eason($\mathcal{P}$), where $\mathcal{P} = ^{(\mathrm{R},\ \mathrm{F}}, \pi)$

1: $k \cdots = 0$; G^0 是 R 的空解释图;
2: **do**
3: $k := k + 1$;
4: Compute G^k starting from G^{k-1} in an $\mathrm{i}_{\mathrm{ncrem}}$ ental fashion
5: **对于每个** 节点 v的深度 k **执行**
6: T $\cdots = \{\mathrm{T}(\alpha, v, \mathrm{F}) \neq \emptyset \mid \alpha$是一个事实$\}$ tset$(v, \mathrm{F}) \cdots = \emptyset$
7: **f或每个h** $\{\tau_1, \cdots\cdots, \tau_n\} \in \mathcal{T}$ **do**
8: **如果** T 中集合的平均大小为 $\geq t$ **则**
9: $Z \cdots = \{$合并$(\tau_1$、……、 $\tau_n)\}$
10: **否则** $Z \cdots = \{\tau_1$、……、 $\tau_n\}$
11: **f或每个h** t棵树 $\tau \in Z$ **d o**
12: **如果** τ 相对于 α 不冗余**那么**
13: **添加** τ到 tset$(v, \mathcal{F})$
14: **如果**tset$(v, \mathrm{F}) = \emptyset$，**则从** G**中移除** v
15: **当**d$(G^k) \neq$ d(G^{k-1})
16: **返回** $G^k(\mathcal{F})$

注: t 是给定的阈值（默认值为 10）。

ε，使得 ε与 τ具有相同的根事实，标签为 与，并且每个根(δ_i) 到根(ε)都有一条边。

在定义 5 中，★表示未发生折叠的情况；† 表示根据定义 4 将多个推导树折叠为一棵，使得新树的根是一个标记为 OR 的节点的情况；最后，‡ 表示 OR 出现在 τ 的祖先节点中的情况。我们通过示例 5 来说明定义 5。

示例 6。假设节点 v_2 中的所有树都已合并为一棵树 ϵ，其根为 $t(a)$，见图 2b，并且节点 v_3 存储一棵单独的推导树 τ，其根是带有“与”标签的事实 $r(a, b_1)$。下面讨论展开 τ 的过程。由于在 τ 的一个非根节点中存在带有 “ 或 ” 标签的节点 $t(a)$，展开 (τ) 将进入情况 ‡，并定义为以下树的集合：计算根(τ) 的两个子节点的展开结果的笛卡尔积，即 ϵ 和仅包含一个节点 $s(a, b_1)$ 的树（后者未在图 2b 中显示）。由于 ϵ 是带有“或”标签的节点，展开(ϵ) 定义为其子节点展开结果的并集，即以 $t(a)$ 为根的 N 棵树（情况 †）。由于其中没有任何一棵包含带有“或”标签的节点，因此根据基本情况（★），它们的展开结果就是树本身。

由于折叠树 τ封装了推导同一事实的多种方式，我们称 τ相对于事实 α是冗余 的，当且仅当 α在展开(τ)中的每个 推导树中至少出现两次。这意味着，如果我们需要检查 τ是否冗余，那么并不总是需要完整计算展开(τ)，因为一旦找到一个非冗余的推导树，我们就可以停止。回到示例 6，

$r\ a$ τ $(, b_1)$在展开()中的一个推导树中出现了两次。然而，τ相对于 $r(a, b_1)$并不冗余，因为其中还包含其他树，在这些树中$r(a, b_1)$只出现一次。

我们在算法2中概述了当某些推导树可以折叠时的推理过程，并将其称为概率折叠推理（PCOReason）。该过程与算法1类似。首先，它计算通过规则实例化可以得到的所有不同推导树（第6行）。然后，算法依次处理所有以同一事实 α为根的树集。第8行的条件决定是否应使用阈值（见下文讨论）折叠 v中的树

。如果应当折叠，则按照定义4折叠每个树集；否则，逐个处理这些树（第11–13行）。

可以采用多种策略来决定是否合并节点中的推导树。在第8行，我们使用一个简单的阈值，将更复杂的策略留待未来工作。我们的策略会考虑具有相同根事实的节点中推导树的平均数量。如果该平均数量至少为 $t = 10$（我们选择了 10 这个值，因为它至少能减少一个数量级），那么就合并该节点中的推导树；否则，我们逐棵存储这些树。

以下结果证明了算法2的正确性。

定理 4。对于每个概率程序 $\mathcal{P}$，PCOReason($\mathcal{P}$) 都是$\mathcal{P}$的谱系 TG。

上述结果源于算法1与算法2之间的密切关系，以及带有 OR 标记节点的冗余概念的正确性。

我们用于压缩谱系的技术与 [28] 中计算溯源电路的技术相似。溯源电路 是由布尔运算符和事实构成的有向无环图，可以紧凑地表示来源信息，从而避免基于来源半环的技术所导致的指数级膨胀 [43]。在 [28],中，作者提出了一种算法，通过在模型计算过程中传播电路来计算来源信息。该技术首先创建一个包含数据库中所有事实的电路。然后，在每一轮次 k，它会使用至少一个在第（$k-1$）轮次中推导出的事实来实例化所有规则——这就是 SNE 引入的约束，见第 2 节。如果无法进行这样的实例化 $\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_N$，计算便会终止。否则，算法将执行以下步骤。如果 α 是首次被推导出来的，就会向电路中添加两个∨节点，一个标注为新变量 X_α，另一个标注为新变量 Y_α。然后，添加一个新的∧节点 u，并添加一条从 X_α 到 u 的边。最后，对于每个 α_i，如果 α_i 是输入事实，就添加一条从 u 到标注为 X_{α_i} 的节点的边；否则，添加一条从 u 到 Y_{α_i} 的边。

示例 7. 图 *3* 展示了根据示例 *5* 中的概率程序计算出的电路。每个节点的标签显示在其顶部。如果某个事实是由规则推导出来的，那么关联变量的上标就是其创建时所在的轮次，即变量 $X^1_{r(a,b_1)}$ 是在第一轮中创建的。

溯源电路中的 ∨（∧）节点与 OR（AND）标签发挥相同的作用。不过，我们的方法折叠谱系的方式有显著不同。首先，我们的方法只折叠存储在单个 TG 节点中的谱系。而在 [28], 中，由于该技术基于 SNE 而非 TG，折叠操作会考虑整个模型。其次，我们的方法具有自适应性，仅在有益时才会启用折叠（见算法2第 8–9 行）。相比之下，在 [28], 中，即使没有必要，也总是会执行该操作。例如，在示例 7 中，每个 r（a、b_i）事实都会创建两个新节点。而图 2 中折叠后的树表示则将每个 r（a、b_i）事实仅存储一次。

6 评估

我们将这一方法实现为一个名为 LTG 的新引擎，并将其性能与实现了

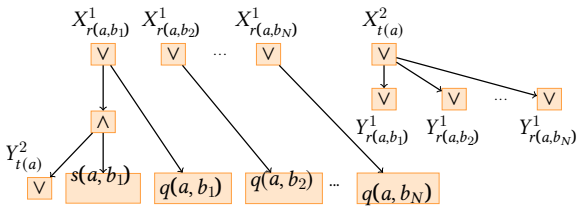


图 3：来自 [28]的示例 5 的溯源电路。

表2：所考察的基准测试。在 Smokers 和 VQAR 中，#DB 和 #DR 分别取决于 N 和每个查询。

	#R	#DB	#DR	#Q
LUBM010	127	1M	1.7M	14
LUBM100	127	12M	18 M	14
DBpedia	9k	29M	33 M	50
Claros	2k	13M	8 M	50
YAGO5	221	1M	1 M	50
YAGO10	396	1M	1 M	76
YAGO15	571	1M	1 M	50
WN18RR5	66	86k	87 k	20
WN18RR10	116	86k	87 k	20
WN18RR15	166	86k	87 k	50
Smokers	5	*	*	110
VQAR	6	*	*	1000

$Tc_\mathcal{P}$ 的 ProbLog [78],、$\Delta Tc_\mathcal{P}$ 的最先进实现 vProbLog，以及 Scallop [49], 这一较新的近似概率推理引擎进行了比较。据我们所知，这些是目前仅有的、用于在可能世界语义下进行推理的最先进引擎。我们分别测试了折叠和不折叠谱系的 LTG，并将这两种情况记作“折叠谱系的 LTGs”和“不折叠谱系的 LTGs”。

为了根据答案的谱系计算其概率，我们考虑了三种最先进的工具：PySDD [23], 来自 [35] 的d-tree编译器，以及 c2d [22]。PySDD 是一种广为人知的 WMC求解器，可以处理析取范式（DNF）中的公式，即 LTG返回的谱系形式。PySDD 将是我们的默认求解器，因为所有竞争对手都采用了它，并且它支持DNF。d-tree编译器是一种性能具有竞争力的替代技术。最后，c2d 是另一种最先进的求解器，在 2021 年模型计数竞赛（https://mccompetition.org/）中排名前三。该求解器要求公式采用 合取范式（CNF）。为了将谱系公式从DNF转换为CNF，我们应用了宽松Tseitin变换 [83]，该变换可以在输入公式大小的多项式时间内完成。

所有实验均在一台配备 Intel i7 CPU 和 94 GiB RAM 的 Ubuntu 16.04 PC 上运行。

6.1 基准测试

我们考虑了来自数据库、概率编程和机器学习领域的基准测试。

• LUBM [46]是一个广受欢迎的 Datalog 基准测试，曾用于评估 ProbLog2、vProbLog 和其他引擎 [60, 77, 80]。我们考察了 LUBM010 和 LUBM100，二者分别包含 1M 和 12M 条事实。我们使用了与竞争对手相同的 127 条规则集，以及 14 个可用查询。

• DBpedia [12] 是由维基百科构建的最知名知识图谱之一。Claros [66] 是一个文化遗产本体。

知识库包括编码了具有 N 个节点和最多 $2 \times N$ 条无向边的随机幂律图的概率数据库。我们考虑了介于 10 和20之间的 N，如 [78] 和现有的 110 个查询中所述。我们将最大推理深度限制为四步和五步，与 [78] 中相同。

• Smokers [30] 是人工智能社区中广受欢迎的知识库。

• VQAR [49] 已被提出用于视觉问答领域中的基于规则的推理。该基准测试提供了5000多对查询和概率程序。每个程序包括：(i) 将图像上的神经网络预测转换为关系形式后获得的不确定事实，以及 (ii) 取自CRIC本体 [40]的规则和事实。我们考虑了需要最多推理步骤的1000个查询。VQAR具有挑战性，因为推导的数量呈组合爆炸式增长。

规则挖掘基准测试。 我们还考虑了使用 AnyBurl [57], 这一最先进的知识图谱补全技术挖掘规则的场景。该技术优于此前的知识图谱嵌入技术（例如 ComplEx [54], ）以及其他规则挖掘技术。AnyBurl 挖掘出的每条规则都会根据其在数据中的支持度赋予一个置信度。为挖掘规则，我们考虑了机器学习社区常用的两个知识图谱：YAGO3 [55] （下文称为 YAGO）和 WN18RR [26]。对于每个知识图谱，我们分别为每个谓词选取置信度最高的前 5、10 和 15 条规则，从而构建了三个不同的基准测试。YAGO 和 WN18RR 均提供训练、验证和测试知识图谱三元组集合。训练和验证三元组用于挖掘规则，测试三元组则用于推理时。

每个基准测试 B 都是构建不同场景的基础。每个场景由 B 中的数据库、规则和查询构成。我们用基准测试名称加上可能的参数来表示场景，例如，Smokers4 使用 Smokers 知识库，并将最大推理深度设为 4。只有 Smokers 和 VQAR 定义了概率函数 π。对于其余基准测试，我们通过为每条事实赋予一个位于 $(0, 1]$ 内的随机数来实现 π。这与 ProbLog2 和 vProbLog [78]采用的方法相同。对于未提供查询的基准测试，我们使用[50]中的方法创建了包含 1、2、3 和 4 个原子的查询。回答这些查询所需的推理步骤数各不相同。表2报告了所有场景的统计数据。#R、#DB、#DR 和 #Q 分别表示规则数、事实数、不同事实推导数和查询总数。

6.2 问答方法

为了与 ProbLog2 和 vProbLog 的评估保持一致，我们应-用了魔术集 (MS) 转换 [5, 8, 11]。MS 是一种数据库技术，给定一个查询和一个非概率程序 P，它会重写 R 中的规则，使得对重写后规则的自底向上求值能够模拟使用 R 对查询进行的自顶向下求值。Tsamoura 等人 [78] 已证明，MS 也支持概-率程序。

对于 VQAR 场景以外的所有场景，我们采用如下实验方法。对于每个包含程序 $\mathrm{P} = (\mathrm{R}, \mathrm{F}, \pi)$ 和查询 Q 的场景，我们将 P 转换为一个新程序 P^Q，并使用 MS。在 ProbLog2、vProbLog 和 Scallop 中，查询回答首先计算 P^Q 的最小参数化模型 M（推理步骤）。然后，对于 M 中的每个查询事实 $Q(\mathrm{c})$，我们计算其关联公式的概率（概率计算步骤）。对于 LTG，查询回答首先计算 P^Q 的谱系 TGG（推理步骤），然后计算 $G(\mathrm{F})$ 中每个 $Q(\mathrm{c})$ 的谱系（谱系收集步骤），最后计算该谱系的概率（概率计算步骤）。对于 VQAR，我们没有应用 MS，而是直接使用 Scallop 作者提出的查询 [49]。

LTG、ProbLog2 和 vProbLog是精确的概率推理引擎，而Scallop 是一种近似引擎，只保留每个派生事实的前 k 个解释。为了确保公平比较，我们对Scallop 进行了配置，使其计算出的解释数量尽可能接近其他精确引擎计算出的数量。为此，我们将 $k = 30$ 设置为默认值，这是 Scallop能够回答大多数查询的最高可能值（对于更高的值，计算通常会导致内存不足）。请注意，即使使用 $k = 30$，Scallop 在某些情况下仍然会进行近似，因此具有一定优势。我们使用Scallop(k) 表示将 Scallop应用于特定的 k。我们观察到，当查询回答过程终止时，通常会在几分钟内完成。因此，我们设置了 30 分钟的超时时限，以便在无需等待太久的情况下尽可能回答更多查询。

6.3 结果

对于 LUBM，我们比较 LTG 与所有其他引擎。对于 DBpedia、Claros、Smokers、YAGO 和 WN18RR，比较中不包括 ProbLog2 和 Scallop。对于 ProbLog2，正如 [78] 也观察到的那样，它在 LUBM010 上的性能过低，因此不再纳入考量。对于 Scallop，它要么不支持基准测试中的某些规则，要么数据量过大而无法加载（Scallop 的作者将可扩展性列为未来工作方向 [49]）。

对于 VQAR，我们仅比较折叠谱系的 LTG 和 Scallop：由于推导的组合爆炸，不折叠谱系的 LTG 和 vProbLog 均无法计算最小参数化模型。尽管存在这种组合爆炸，LTG 仍是唯一能够计算完整最小参数化模型的技术。

请注意，计算完整谱系并不意味着总能计算出其精确概率，因为该问题是 #P 难的 [84]。对于此类情况，可以采用近似方法：既可以预先通过缩减谱系大小来实现（Scallop），也可以在收集完整谱系后再进行近似，如 [41, 62, 84] 所示。对谱系概率进行近似是一个正交问题。因此，我们将这类技术与 LTG 的集成留作未来工作，重点关注那些可以使用 PySDD 精确计算答案概率的查询。在 VQAR 中，此类查询有 417/1000 个。对于其余查询，PySDD 要么无法完成计算，要么谱系过大（$> 1M$ 个析取项），以至于尽管在某些情况下可以完整计算，但所需时间会超出超时时限。

实验结果概览。表 3 报告了使用所有引擎回答 LUBM010（左侧）和LUBM100（右侧）中查询的总计运行时间 。对于 LTG，总计运行时间是推理、谱系收集和概率计算时间之和。图4展示了vProbLog 和 LTG 中上述步骤的细分。谱系收集与

.....无关

表 3：使用 ProbLog2 (P)、Scallop (S)、vProbLog(vP) 和 LTG (L) 回答 LUBM010 与 LUBM100 中查询所需的总时间（默认单位为毫秒）。概率通过 PySDD (SDD)、d-tree 和 c2d 计算。阴影单元格显示了最佳时间。

	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}	Q_1	Q_2	Q_3	Q_4	Q_5	Q_6	Q_7	Q_8	Q_9	Q_{10}	Q_{11}	Q_{12}	Q_{13}	Q_{14}
P+SDD	59	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	78	北美 150		NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA	NA
S(30)+SDD	1.3s	NA	729	NA	4.5s	817s	6s	NA	NA	NA	63	165s	30s	326	15.5	NA	8.9s	NA	NA	NA	NA	NA	NA	NA	NA	372	NA	NA	3.3s
vP+SDD	587	7.2s	306	5.6s	13.6s	NA	6.3s	NA	NA	1.3s	2s	17.3秒	12.4秒	3.1s	7.3s	NA	2.5s	NA	NA	NA	NA	NA	NA	NA	2s	NA	NA	38.7秒
L 不含+SDD	57	420	38	1.1s	1.3s	NA	353	35.1秒	348s	187	7	10.6秒	541	337	647	52s	455	2.4s	4.7s	NA	2s	51.8 秒	NA	1.7s	31	12.7秒	6.1s	4.9s
L 带有 +SDD	49	383	38	175	365	NA	315	21.8s	174s	162	5	387	176	273	617	46.1秒	444	1.5s	3.7s	NA	1.9s	71.4 秒	NA	1.6s	21	1.6s	2.8s	6s
L w/+d-tree	49	676	40	461	595	NA	4.9s	668s	108s	1.5s	6	1s	206	273	617	42s	411	1.7s	2.7s	NA	6.3s	658s	NA	2.9s	21	2s	2.4s	6s
L w/+c2d	49	41s	316	3.9s	62s	NA	27s	NA	NA	2.4s	6	13s	6.2s	273	617	NA	1s	7.4s	113s	NA	32s	NA	NA	4.2s	21	16s	16s	6s

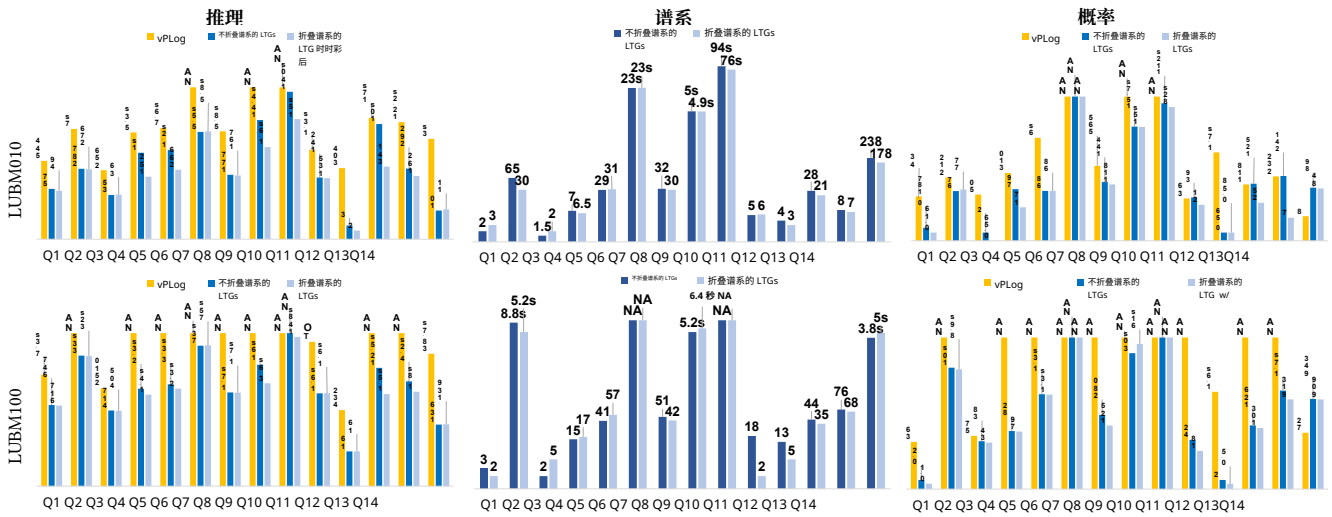


图 4：回答 LUBM 查询的运行时间明细（毫秒）。

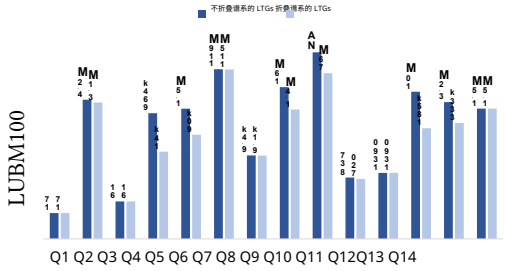


图5：LUBM 查询的推导数量。

表 4：推理过程中合并谱系的绝对（毫秒）和相对运行时间开销。

	LUBM010	LUBM100		LUBM010	LUBM100
Q1	0.009（0.01%）	0.007（0.001%）	Q8	28（1.7%）	27（0.6%）
Q2	1.6（0.5%）	54 (0.1%)	Q9	115 (0.7%)	1165（0.7%）
Q3	0.02 (0.05%)	0.01（0.002%）	Q10	151（0.1%）	0.2（0.01%）
Q4	0.8（0.4%）	0.7（0.04%）	Q11	0.03（0.6%）	0.04（0.1%）
Q5	2.8（1%）	3（0.1%）	Q12	6.3（1.5%）	6（0.3%）
Q6	204（3.4%）	2239 (2.8%)	Q13	1.6 (0.8%)	17 (0.8%)
Q7	183 (0.1%)	0.3 (0.01%)	Q14	0.9 (6.9%)	10（6.5%）

vProbLog，因为 ΔTcp 不需要这一步。图5显示了使用 LTG 回答 LUBM 查询时产生的推导数量。推导数量可以粗略估算

表5：计算 LUBM010 每个查询答案的概率所需的平均运行时间（毫秒）和标准差。

	vProbLog + PySDD	折叠谱系的 LTG + PySDD	折叠谱系的 LTG + d-tree	折叠谱系的 LTG + c2d
Q1	7.5 ±0.1	0.001 ±0.0008	0.001 ±0.0007	0.002 ±0.0007
Q2	7.3 ± 0.2	3.6 ±1.7	13 ±27	1.5秒 ±389
Q3	7.4 ±0.08	0.06/0.1	0.2 ±0.08	78 ±9
Q4	7.4 ± 0.07	0.5 ±0.7	8.9 ±49	146 ±182
Q5	7 ±0.08	0.1±0.1	0.4 ±2	119 ±80
Q6	NA	NA	NA	NA
Q7	7.1 ±0.07	1.9 ±1.3	70 ±163	443 ±164
Q8	NA	2 ±1.3	84 ±110	521 ±147
Q9	NA	33 ±39	6 ±10	TO
Q10	7.5 ±0.09	6 ±2	358 ±200	614 ±163
Q11	6.9 ± 0.08	0.0009 ±0.0003	0.0008 ±0.0007	0.002 ±0.0005
Q12	7.5 ± 0.09	2.6 ±1	44 ±36	980 ±59
Q13	6.8 ± 0.04	0.2 ±0.2	1 ±1	234 ±75
Q14	6.9 ±0.3	0.0008 ±0.0005	0.0007 ±0.001	0.002 ±0.0008

每个查询的难度，且与具体实现无关。表 3 以及图4和图5中的“NA”表示超时时限或内存不足（稍后会给出详细分类）。表4显示了在 LUBM 上折叠谱系所需的开销（包括绝对开销，单位为毫秒；以及相对于总推理时的开销），而表5报告了使用不同技术计算查询答案概率的平均运行时间（单位为毫秒）。

图 6 展示了 DBpedia、Claros、Smokers、YAGO 和 WN18RR 场景中的推理、概率计算以及查询回答总耗时。“推导”下方的数值

表6：回答基准测试查询所需 RAM 峰值用量（GB）的最小值和最大值，以及 OOM 和 TO 查询的数量。

	vProbLog		不折叠谱系的 LTG		折叠谱系的 LTG	
	最小值/最大值	内存不足/超时	最小值/最大值	内存不足/超时	最小值/最大值	内存不足/超时
L10	11/11	1/2	1.7/19	1/0	1.9/11	1/0
L100	13/14	2/12	1.8/5.7	2/0	1.9/4.8	2/0
D	30/38	3/0	2/4.9	3/0	1.9/2.5	3/0
C	19/20	3/0	2.5/6.1	3/0	2.4/5.2	3/0
Y5	11/11	12/8	1.8/1.8	12/0	1.8/1.8	11/0
Y10	11/11	30/19	1.8/1.8	25/0	1.8/1.8	25/0
Y15	11/11	12/18	2.4/2.4	12/8	1.7/1.7	12/8
W5	11/11	0/0	1.8/1.8	0/0	1.8/1.8	0/0
W10	11/11	0/0	1.8/1.9	0/0	1.8/1.8	0/0
W15	11/11	0/0	1.9/1.9	0/0	1.8/1.8	0/0
S4	11/11	0/0	1.7/1.7	0/0	1.7/17	0/0
S5	11/11	0/0	1.8/17	0/0	1.7/19	0/0
V		1000/0		1000/0	1.4/24	560/23

表示 LTG 的推导数量。由于每个场景中的查询数量很多，我们采用了箱线图。箱线图汇总了所有在 30 分钟内完成评估的查询耗时。对于 Smokers，k 表示最大推理深度，可以是四或五。相比之下，YAGO 和 WN18RR 场景中的不同 k 表示每个谓词保留的置信度最高的规则数量。表6列出了评估未能在超时时限内完成（“TO”列）或因内存不足而中止（“OOM”列）的查询数量（各场景的查询数量见表2）。对于成功回答的查询，我们记录了引擎使用的峰值 RAM。表6报告了各场景中得到的最小值和最大值（以场景名称首字母表示，例如“L”表示 LUBM），以展示最佳和最差情况下的内存需求。

图7a、7b和7c报告了从417个 VQAR 查询中收集的结果，其中 LTG 能够计算精确答案。为了展示近似对运行时间的影响，图7a报告了 Scallop(1)（S(1)）、Scallop(20)（S(20)）和 LTG（LTG 总计）所需总运行时间的比较。对于 LTG，图中还报告了推理（Reas.）、谱系收集（Lin.）和概率计算（Prob.）所需运行时间的细分。所有时间均以毫秒为单位。由于我们的引擎支持精确的查询回答，我们还评估了近似对答案概率的影响，即评估近似概率与实际概率的接近程度。为此，图7b报告了 Scallop(1) 和 Scallop(20) 计算出的答案的相对概率误差。答案 α 的相对概率误差计算为 $(\epsilon_\alpha - \hat{\epsilon}_\alpha)/\epsilon_\alpha$，其中 ϵ_α 表示精确概率（由 LTG 计算），$\hat{\epsilon}_\alpha$表示近似值。在此实验中，417个查询产生了5949个答案。图7b根据相对误差对答案进行分组，并报告每组中的答案总数，即对于 S(1)，有168个答案的误差落在 $[0, 10\%)$ 内。最后，为了提供一些示例性证据，我们针对不同 k选择了 Scallop 回答耗时最长的5/417个查询。表7c展示了这些查询分别使用 Scallop 和 LTG 所需的总运行时间，以及其答案的最高概率。

6.4 主要结论

C1：*LTG* 在运行时间方面优于现有技术。 表 3 和图 6 表明，使用 LTG 进行查询回答比其更快于与其他引擎相比。例如，采用 LTG 后，DBpedia 和 Claros 的最大总运行时间分别从采用 vProbLog 时的 195秒和 26秒降至 129秒和 11秒。采用 LTG 后，DBpedia 和 Claros 的平均总运行时间分别从采用 vProbLog 时的 14秒和 8秒降至 6.6秒和 1.4秒。YAGO 和 WN18RR 的提升幅度更大：YAGO15 的平均运行时间从 0.7秒降至 0.1秒，WN18RR15 则从 0.2秒降至 0.01秒。

更重要的是，LTG 可能决定查询究竟能否得到回答。例如，LTG 能够成功回答 LUBM010 中的 14 个查询中的 13 个（其中大多数只需数秒），以及 LUBM100 中的 14 个查询中的 12 个。在 LUBM010 中，对于Q_6，LTG 成功完成了推理和谱系收集，但 PySDD 则内存不足。在 LUBM100 中，使用 LTG 处理 Q_6和 Q_9 时，内存不足发生在谱系收集阶段；推理已成功完成。与第二快的精确引擎 vProbLog 相比，除双方都超时的查询外，LTG 处理所有查询的速度都更快，提速幅度超过一个数量级，见 Q_1 和 Q_5。表 3 还显示，LTG 通常比 Scallop(30) 更快，除了 LUBM010 中的 Q_6（Scallop 在 817 秒内返回近似答案）以及 LUBM100 中的 Q_{14} 之外，LTG 在所有情况下都胜过 Scallop。对于 Q_6，LTG 执行的概率计算本身开销很大，因此采用近似方法的 Scallop 运行得更快。对于 Q_{14}，该查询几乎不需要推理，而 vProbLog 的开销更低。

C2：折叠谱系可以显著提升性能。 例如，考虑 LUBM010 中的查询 Q_{12}。使用不折叠谱系的 LTG 需要 10.6 秒才能回答 Q_{12}，而使用折叠谱系的 LTG 只需 387 毫秒。这是因为，对于 LUBM010，使用不折叠谱系的 LTG 进行推理需要 10 秒，而使用折叠谱系的 LTG 只需 341 毫秒，见图 4。在其他查询中也观察到了显著的性能提升，例如 LUBM010 中的 Q_4 和 LUBM010 中的 Q_2。总体而言，在大多数情况下，折叠谱系的 LTG 至少比不折叠谱系的 LTG 快 25 %。在 VQAR 查询中观察到的差异最大，因为折叠谱系使我们能够为所有查询计算完整的最小参数化模型。推理时得到改善的原因是推导数量大幅减少。例如，LUBM010 中的 Q_{12} 使用不折叠谱系的 LTG 涉及 1000 万个推导，见图 5。相比之下，同一查询使用折叠谱系的 LTG 仅涉及 18.5 万个推导。在 DBpedia 和 Smokers 中，推导数量也显著减少，而在其他情况下则大致保持不变，见图 6。

通常，折叠谱系操作带来的开销可以忽略不计（即低于 1%，见表 4）。不过，也有少数情况并非如此，例如 Q_{12} 和 Q_{14}。对于 Q_{12}，尽管开销不可忽略，但运行时间上的改善足以抵消折叠的成本。然而，Q_{14}并非如此，因为该查询触发的推理不足以证明折叠操作是合理的，因此折叠不再有益。

C3：收集谱系的运行开销很小。在大多数场景中，与推理所需的时间相比，计算谱系的开销相对较小。例如，在 DBpedia 和 Claros 中，收集谱系所需的最长时间分别为 1581 毫秒和 1533 毫秒。由于 LTG 能够更高效地推理，付出这点开销是值得的，因为这样可以大幅缩短总运行时间。在 LUBM 中，我们发现有两种情况的谱系收集成本高得难以承受：Q_6 和 Q_9 在 LUBM100 中。这是

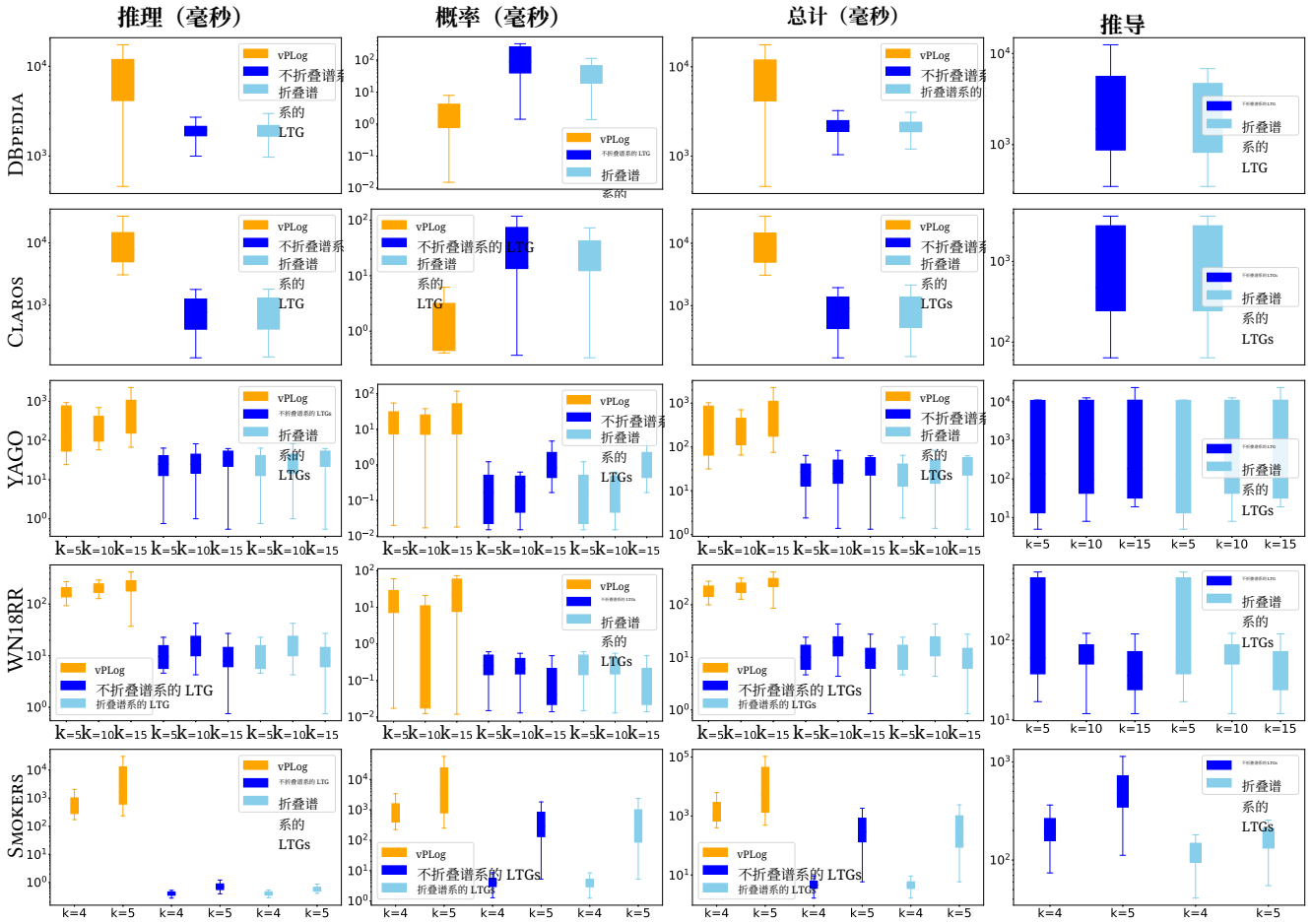
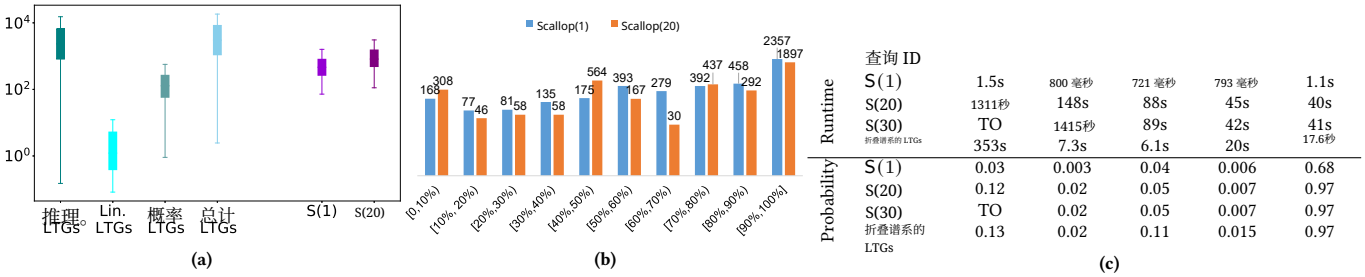


图 6：使用vProbLog 和 LTG 时，不同场景下回答查询所需的时间（毫秒）以及推导总数。



	查询 ID					
Runtime	S(1)	1.5s	800 毫秒	721 毫秒	793 毫秒	1.1s
	S(20)	1311秒	148s	88s	45s	40s
	S(30)	TO	1415秒	89s	42s	41s
	折叠谱系的 LTGs	353s	7.3s	6.1s	20s	17.6秒
Probability	S(1)	0.03	0.003	0.04	0.006	0.68
	S(20)	0.12	0.02	0.05	0.007	0.97
	S(30)	TO	0.02	0.05	0.007	0.97
	折叠谱系的 LTGs	0.13	0.02	0.11	0.015	0.97

(c)

图7：与 LTG能够精确回答的 417 个VQAR 查询相关的结果。（a）折叠谱系的 LTG 的运行时间细分，以及Scallop中以毫秒为单位的总计运行时间；（b）进行近似时得到的相对概率误差；（c）五个查询的轶事证据。

因为相关答案数量过于庞大，谱系收集的成本抵消了推理运行时间的缩短。*C4*：在内存方面，*LTG* 优于以往的方法。如表6所示，在 LUBM 场景中，LTG 的内存效率最高可达 vProbLog 的四倍；改进幅度超过六

在 YAGO 和 WN18RR 场景中，所需时间更短。这是因为 LTG 并未完全物化这些树，而是存储指向父树的指针（结构共享）。折叠推导树的操作进一步降低了内存消耗。再看

表6，我们注意到，最佳情况下，最大 RAM 使用量几乎减半（DBpedia，4953MB 对比 2550MB）。唯一需要更多 RAM 的场景是 Smokers5（S5），因为 vProbLog 使用的基于列的数据结构占用空间更少。对于 LUBM010 和不折叠谱系的 LTGs，最大 RAM 消耗较高（不折叠谱系的 LTGs 为 19GM，而 vProbLog 为 11GB），这是由于 Q_9，一个 vProbLog 无法回答的查询。尽管 LTG 总体上更节省内存，但仍有一些查询会因 RAM 不足而无法处理。原因在于当前问题在最坏情况下不可处理。一个例子是来自 LUBM010 的 Q_6：推理和谱系收集步骤均成功完成，但 PySDD 执行概率计算时内存不足。我们工作的一个重要贡献是显著减少了此类情况的数量，见 VQAR。此外，在最具挑战性的情况下，LTG 使用 24GB RAM 进行推理，这表明 LTG 无需昂贵的硬件即可支持复杂场景。

C5：*LTG* 可以与不同的概率计算技术结合使用。表 3 显示，即使采用不同于 PySDD 的概率计算工具，LTG 也能达到最先进的性能：与 c2d 结合时，折叠谱系的 LTG 在 LUBM010 的 14 个查询中有 5 个、在 LUBM100 的 14 个查询中有 9 个优于 vProbLog；使用 d-tree 时，折叠谱系的 LTG 几乎总是比 vProbLog 快。表5 显示，就运行时间而言，PySDD 通常是效率最高的库，而 c2d 的单个答案运行时间最长。这是因为通过 Tseitin 变换将 DNF 转换为 CNF，会在谱系公式中不同的析取项之间产生相互依赖关系，使得 c2d 所需的公式分解变得更加困难。另一个值得指出的现象是，与 vProbLog 搭配相比，PySDD 与 LTG 搭配时表现要好得多。原因在于，PySDD 会将谱系转换为一种称为 vtree 的内部形式 [63]。这种转换的成本取决于公式的结构；也就是说，两个公式（一个由 LTG 返回，另一个由 vProbLog 返回）可能逻辑等价，但将它们转换为 vtree 的成本可能不同。这解释了运行时间上的差异，这一现象在 [78] 中也有观察到。

C6：*LTG* 具有与近似技术相当的竞争力。毫不意外，仅保留排名靠前的 k 个证明来近似推理会带来更短的运行时间（图7a）。然而，这样做返回的概率可能与实际概率相去甚远。如果采用激进的近似方法，例如使用 Scallop(1)，差异可能会非常显著。例如，从图7b 可以看到，5949 个答案中有 2357 个的相对误差大于90%。图 7c 给出了几个说明性示例：查询 2322829_40与 2416754_49 的答案，其概率至少比实际概率低两倍。若要减少近似误差，就需要增大 k，例如运行 Scallop(20) 和 Scallop(30)。然而，这样做会使 Scallop 的运行时间增加到不再适合采用近似的程度。

7 相关工作

包括 ICL [65], PRISM [73], 马尔可夫逻辑网络 [71], 与 PSL [4]在内的几种方法都能在不确定性下进行推理。ICL 与 PRISM 仅支持度这样的规则：同一个谓词不能同时出现在其前提和结论中。马尔可夫逻辑网络与 PSL 并非基于逻辑编程而非一阶逻辑。因此，它们不支持非基项递归规则，因为一阶逻辑无法指定传递关系 [42]的闭包。随机逻辑程序 [16], TensorLog [15], 和概率 Datalog [38, 39] 不支持可能世界语义。Bárány 等人 2017年提出了另一种概率版 Datalog，称为 PPDL [6]。与我们的工作不同，PPDL 的语义使用马尔可夫链定义。我们不了解任何 PPDL 引擎。

一些方法通过计算谱系的子集来降低计算完整谱系的成本。第一个 ProbLog 引擎 [25]实现了迭代加深；ProbLog2 实现了 k-best [47]与 k-最优 [70] 近似方法；Scallop 为每个事实保留 k 个最可能的解释，从而实现了最先进的性能 [49]。这些方法与 LTG 的主要区别在于，后者执行精确推理。我们的评估表明，尽管 LTG 通常需要比近似方法做更多工作，但在某些情况下，LTG 仍然显著更高效。

许多近似技术致力于解决 DNF 概率计算问题 [19, 62, 68]。Van den Heuvel 等人[84] 最近提出了一种随时近似技术，该技术基于“解离”界构建 [41]。将此类技术整合到 LTG 中是一个有趣的方向，因为当谱系过大、无法进一步处理时，这可以扩大其适用范围。

关于概率数据库上的查询回答的研究为我们带来了丰硕的成果，尤其是在复杂查询的可处理性方面，例如 [18, 20],，以及近似方法方面，例如 [44]。近期的工作包括为查询寻找解释，例如 [45],，以及在约束条件下进行查询 [36]。MystiQ [17] 和 MayBMS [3]等系统提出了对 PostgreSQL 等数据库管理系统的扩展，以支持概率数据库的语义；而 PrDB [75] 则引入了利用图模型扩展数据库的技术。在 [32] 和 [33], 中，Dylla 等人分别研究了时态概率数据库上的查询回答问题，并提出了针对概率数据库的 top-k 查询回答技术。与我们的工作不同，上述研究方向侧重于支持 SQL 查询，而非视图重构之外的基于规则的推理。

LTG 与 VLog [80],、RDFox [60] 和 Vadalog [9]等高性能 Datalog 引擎有关。研究表明，（非概率）基于 TG 的推理在运行时间和内存消耗方面优于上述引擎 [77]。此外，基于 TG 的推理能够自然地维护推导溯源，且不会因引入 TG 而产生额外开销。上述引擎均不易以这种方式扩展，因为它们都实现了 chase [10],，这会将事实与推导出这些事实的规则“断开关联”。

ProbLog2、vProbLog 和 Scallop 与来源半环密切相关。Green 等人使用半环为 Datalog 定义了来源信息，并支持可能世界语义 [43]。 [43] 与上述引擎的区别在于，后者利用自底向上 Datalog 求值的思想来缩短运行时间。 [67] 中的作者提出了一种 SNE 自底向上方法，用于近似特定类别半环的来源信息，以找到权重最佳的推导。秉持同样的思路，[27] 提出了近似溯源计算技术，而 [21] 则为涉及否定和不动点运算的非常通用的逻辑语言开发了半环溯源。

8 结论

我们提出了一种新的可扩展技术，用于在概率数据库上进行概率规则推理。该技术利用 TG 的拓扑结构和结构共享，计算紧凑的概率模型。实验表明，我们的引擎在运行时间和内存消耗方面均优于以往的方法，优势往往十分显著；有时，它甚至能决定查询是能在几秒内得到回答，还是完全无法回答。未来的研究方向包括扩展 LTG，以便在 KG 嵌入模型（例如 [37]）上进行推理。一个很有前景的方向是将 LTG 与近似技术相结合，只计算部分谱系 [25, 47, 48, 61, 64, 70],；或者结合通过机器学习（即强化学习）来引导证明计算的技术 [51]。

参考文献

[1] Serge Abiteboul、Richard Hull 和 Victor Vianu。1995。《数据库基础》。Addison-Wesley。[2] Somak Aditya、Yezhou Yang 和 Chitta Baral。2019。在图像理解中整合知识与推理。载于 IJCAI。6252–6259。[3] Lyublena Antova、Thomas Jansen、Christoph Koch 和 Dan Olteanu。2008。不确定数据的快速简易关系处理。载于 ICDE。983–992。[4] Stephen H. Bach、Matthias Broecheler、Bert Huang 和 Lise Getoor。2017。铰链损失马尔可夫随机场与概率软逻辑。《机器学习研究杂志》18（2017），109:1–109:67。[5] François Bancilhon、David Maier、Yehoshua Sagiv 和 Jeffrey D. Ullman。1986。魔术集及实现逻辑程序的其他奇特方法。载于 PODS。1–15。[6] Vince Bárány、Balder ten Cate、Benny Kimelfeld、Dan Olteanu 和 Zografoula Vagena。2017。基于 Datalog 的声明式概率编程。《ACM 数据库系统汇刊》42，4（2017），22:1–22:35。[7] Pablo Barceló 和 Reinhard Pichler（编）。2012。学术界与工业界中的 Datalog——第二届国际研讨会。《计算机科学讲义》，第7494卷。Springer。[8] Catriel Beeri 和 Raghu Ramakrishnan。1991。论魔术的力量。《逻辑编程杂志》10（1991），255–299。[9] L. Bellomarini、E. Sallinger 和 G. Gottlob。2018。Vadalog 系统：面向知识图谱的基于 Datalog 的推理。PVLDB 11，9（2018），975–987。[10] Michael Benedikt、George Konstantinidis、Giansalvatore Mecca、Boris Motik、Paolo Papotti、Donatello Santoro 和 Efthymia Tsamoura。2017。对 chase 进行基准测试。载于 PODS。37–52。[11] Michael Benedikt、Boris Motik 和 Efthymia Tsamoura。2018。带等式的存在规则的目标驱动查询回答。载于 AAAI。1761–1770。[12] C. Bizer、J. Lehmann、G. Kobilarov、S. Auer、C. Becker、R. Cyganiak 和 S. Hellman。2009。DBpedia——Web 数据的结晶点。《网络语义学杂志》7，3（2009），154–165。[13] Antoine Bosselut、Hannah Rashkin、Maarten Sap、Chaitanya Malaviya、Asli Celikyilmaz 和 Yejin Choi。2019。COMET：用于自动构建知识图谱的常识变换器。载于 ACL。4762–4779。[14] Ashok K. Chandra 和 Philip M. Merlin。1977。关系数据库中合取查询的最优实现。载于 STOC。77–90。[15] William W. Cohen、Fan Yang 和 Kathryn Mazaitis。2020。TensorLog：一种使用深度学习基础设施实现的概率数据库。《人工智能研究杂志》67（2020），285–325。[16] James Cussens。2000。随机逻辑程序：采样、推理与应用。载于 UAI。115–122。[17] Nilesh Dalvi、Christopher Ré 和 Dan Suciu。2009。概率数据库：污垢中的钻石。《ACM 通讯》52，7（2009），86–94。[18] Nilesh N. Dalvi 和 Dan Suciu。2007。概率结构上合取查询的二分性。载于 PODS。293–302。[19] Nilesh N. Dalvi 和 Dan Suciu。2007。概率数据库上的高效查询求值。《VLDB 杂志》16，4（2007），523–544。[20] Nilesh N. Dalvi 和 Dan Suciu。2012。合取查询并集的概率推理二分性。《ACM 杂志》59，6（2012），30:1–30:87。[21] Katrin M. Dannert、Erich Grädel、Matthias Naaf 和 Val Tannen。2021。定点逻辑的半环溯源。载于 CSL，第183卷。17:1–17:22。[22] Adnan Darwiche。2004。将 CNF 编译为可分解否定范式的新进展。载于 ECAI。318–322。[23] Adnan Darwiche。2011。SDD：命题知识库的一种新型规范表示。载于 IJCAI。819–826。[24] Luc De Raedt 和 Angelika Kimmig。2015。概率（逻辑）编程概念。《机器学习》100，1（2015），5–47。[25] Luc De Raedt、Angelika Kimmig 和 Hannu Toivonen。2007年。ProbLog：一种概率 Prolog 及其在链接发现中的应用。载于 IJCAI。2462–2467。[26] Tim Dettmers、Pasquale Minervini、Pontus Stenetorp 和 Sebastian Riedel。2018年。卷积二维知识图谱嵌入。载于 AAAI。1811–1818。[27] Daniel Deutch、Amir Gilad 和 Yuval Moskovitch。2018年。使用 top-k 查询高效追踪 Datalog 来源。VLDB Journal 27，2（2018），245–269。[28] Daniel Deutch、Tova Milo、Sudeepa Roy 和 Val Tannen。2014年。用于 Datalog 来源的电路。载于 ICDT。201–212。[29] A. Deutsch、A. Nash 和 J. B. Remmel。2008年。重新审视 chase。载于 PODS。149–158。[30] Pedro Domingos、Stanley Kok、Daniel Lowd、Hoifung Poon、Matthew Richardson 和 Parag Singla。2008年。马尔可夫逻辑。92–117。[31] Xin Luna Dong、Evgeniy Gabrilovich、Geremy Heitz、Wilko Horn、Ni Lao、Kevin Murphy、Thomas Strohmann、Shaohua Sun 和 Wei Zhang。2014年。知识库：一种面向 Web 规模的概率知识融合方法。载于 KDD。601–610。[32] Maximilian Dylla、Iris Miliaraki 和 Martin Theobald。2013年。一种用于信息抽取的时态概率数据库模型。PVLDB 6，14（2013），1810–1821。[33] M. Dylla、I. Miliaraki 和 M. Theobald。2013年。使用非物化视图处理概率数据库中的 top-k 查询。载于 ICDE。122–133。[34] Daan Fierens、Guy Van den Broeck、Joris Renkens、Dimitar Shterionov、Bernd Gutmann、Ingo Thon、Gerda Janssens 和 Luc De Raedt。2015年。使用加权布尔公式对概率逻辑程序进行推理与学习。《逻辑编程理论与实践》（TPLP）15，3（2015），358–401。[35] Robert Fink、Jiewen Huang 和 Dan Olteanu。2013年。概率数据库中的随时近似。VLDB Journal 22，6（2013），823–848。[36] Tal Friedman 和 Guy Van den Broeck。2019年。论受约束的开放世界概率数据库。载于 IJCAI。5722–5729。[37] Tal Friedman 和 Guy Van den Broeck。2020年。向量空间的符号查询：概率数据库与关系嵌入相遇。载于 UAI。1268–1277。[38] Norbert Fuhr。1995年。概率 Datalog——一种用于强大检索方法的逻辑。载于 SIGIR。282–290。[39] Norbert Fuhr。2000年。概率 Datalog：为高级应用实现逻辑信息检索。JASIS 51，2（2000），95–110。[40] Difei Gao、Ruiping Wang、Shiguang Shan 和 Xilin Chen。2019年。从两张图到 N 个问题：一个用于视觉与常识组合推理的视觉问答数据集。CoRR abs/1908.02962（2019）。[41] Wolfgang Gatterbauer 和 Dan Suciu。2014年。布尔函数概率的无条件界。ACM 数据库系统汇刊 39，1（2014）。[42] Erich Grädel。1992年。论传递闭包逻辑。载于《计算机科学逻辑》。Springer Berlin Heidelberg，149–163。[43] Todd J. Green、Grigoris Karvounarakis 和 Val Tannen。2007年。来源半环。载于 PODS。31–40。[44] Eric Gribkoff 和 Dan Suciu。2016年。SlimShot：知识库的数据库内概率推理。PVLDB 9，7（2016），552–563。[45] Eric Gribkoff、Guy Van den Broeck 和 Dan Suciu。2014年。最可能数据库问题。载于 BUDA。1–7。[46] Y. Guo、Z. Pan 和 J. Heflin。2011年。LUBM：OWL 知识库系统的基准测试。《网络语义学杂志》3，2-3（2011）。[47] Bernd Gutmann、Angelika Kimmig、Kristian Kersting 和 Luc De Raedt。2008年。概率数据库中的参数学习：一种最小二乘法方法。载于《机器学习与数据库知识发现》。473–488。[48] Bernd Gutmann、Ingo Thon、Angelika Kimmig、Maurice Bruynooghe 和 Luc De Raedt。2011年。概率编程中逻辑推理的奥妙。《逻辑编程理论与实践》11，4-5（2011），663–680。[49] Jiani Huang、Ziyang Li、Binghong Chen、Karan Samel、Mayur Naik、Le Song 和 Xujie Si。2021年。Scallop：从概率演绎数据库到可扩展的可微推理。载于 NeurIPS。25134–25145。[50] Unmesh Joshi、Ceriel J. H. Jacobs 和 Jacopo Urbani。2020年。改写还是不改写？基于机器学习的算法选择，用于知识图谱上的 Datalog 查询回答。载于 ECAI。792–799。[51] Cezary Kaliszyk、Josef Urban、Henryk Michalewski 和 Miroslav Olsák。2018年。定理证明的强化学习。载于 NeurIPS。8836–8847。[52] Nodira Khoussainova、Magdalena Balazinska 和 Dan Suciu。2008年。从 RFID 数据中进行概率事件抽取。载于 ICDE。1480–1482。[53] A. Kimmig、B. Demoen、L. De Raedt、V. Santos Costa 和 R. Rocha。2011年。论概率逻辑编程语言 ProbLog 的实现。《逻辑编程理论与实践》11，2-3（2011），235–262。[54] Timothée Lacroix、Nicolas Usunier 和 Guillaume Obozinski。2018年。用于知识库补全的典范张量分解。载于 ICML。2869–2878。[55] Farzaneh Mahdisoltani、Joanna Biega 和 Fabian Suchanek。2014年。Yago3：一个来自多语言维基百科的知识库。载于 CIDR。[56] Robin Manhaeve、Sebastijan Dumancic、Angelika Kimmig、Thomas Demeester 和 Luc De Raedt。2018年。DeepProbLog：神经概率逻辑编程。载于 NeurIPS。3749–3759。[57] Christian Meilicke、Melisachew Wudage Chekol、Daniel Ruffinelli 和 Heiner Stuckenschmidt。2019年。用于知识图谱的随时自底向上规则学习

补全。载于 IJCAI。3137–3143。[58] Boris Motik、Yavor Nenov、Robert Piro 和 Ian Horrocks。2015年。通过改写处理 owl:sameAs。载于 AAAI。231–237。[59] W. E. Moustafa、V. Papavasileiou、K. Yocum 和 A. Deutsch。2016年。Datalography：在图处理系统上扩展 Datalog 图分析。载于 IEEE 国际大数据会议。56–65。[60] Yavor Nenov、Robert Piro、Boris Motik、Ian Horrocks、Zhe Wu 和 Jay Banerjee。2015年。RDFox：一种高度可扩展的 RDF 存储库。载于 ISWC。3–20。[61] Davide Nitti、Tinne De Laet 和 Luc De Raedt。2014年。关系对象跟踪与学习。载于 ICRA。935–942。[62] Dan Olteanu、Jiewen Huang 和 Christoph Koch。2010年。概率数据库中的近似置信度计算。载于 ICDE。145–156。[63] Knot Pipatsrisawat 和 Adnan Darwiche。2008年。基于结构化可分解性的新编译语言。载于 AAAI。517–522。[64] David Poole。1992年。逻辑编程、溯因与概率。载于 FGCS。530–538。[65] David Poole。2008年。独立选择逻辑及其后续发展。载于《概率归纳逻辑编程》。222–243。[66] Sebastian Rahtz、Alexander Dutton、Donna Kurtz、Graham Klyne、Andrew Zisserman 和 Relja Arandjelovic。2011年。CLAROS——携手开创过去的未来。载于 DH。斯坦福大学图书馆，355–357。[67] Yann Ramusat、Silviu Maniu 和 Pierre Senellart。2021年。一种用于计算 Datalog 来源的实用动态规划方法。CoRR abs/2112.01132（2021）。[68] Christopher Ré 和 Dan Suciu。2008年。概率数据库的近似谱系。载于 VLDB。797–808。[69] Joris Renkens、Angelika Kimmig、Guy Van den Broeck 和 Luc De Raedt。2014年。用于概率逻辑的基于解释的近似加权模型计数。载于 AAAI。2490–2496。[70] Joris Renkens、Guy Van den Broeck 和 Siegfried Nijssen。2012年。k-Optimal：一种用于 ProbLog 的新型近似推理算法。载于《归纳逻辑编程》。33–38。[71] Matthew Richardson 和 Pedro M. Domingos。2006年。马尔可夫逻辑网络。《机器学习》62，1-2（2006），107–136。[72] Dan Roth。1996 82，1 2 1996 273 302。论近似推理的困难性。《人工智能》–（），–。[73] Taisuke Sato。1995年。一种基于分布语义的逻辑程序统计学习方法。载于 ICLP。715–729。[74] Joerg Schoenfisch 和 Heiner Stuckenschmidt。2017年。在概率数据背景下分析真实世界的 SPARQL 查询和基于本体的数据访问。《国际近似推理杂志》90（2017），374–388。[75] Prithviraj Sen、Amol Deshpande 和 Lise Getoor。2009年。PrDB：管理并利用概率数据库中的丰富相关性。《VLDB Journal》18，5（2009），1065–1090。[76] Dan Suciu、Dan Olteanu、Christopher Ré 和 Christoph Koch。2011年。《概率数据库》。Morgan & Claypool 出版社。[77] Efthymia Tsamoura、David Carral、Enrico Malizia 和 Jacopo Urbani。2021年。通过触发图物化知识库。PVLDB 14，6（2021），943–956。[78] Efthymia Tsamoura、Victor Gutiérrez-Basulto 和 Angelika Kimmig。2020年。超越基底化瓶颈：用于概率逻辑程序推理的 Datalog 技术。载于 AAAI。10284–10291。[79] Efthymia Tsamoura、Timothy Hospedales 和 Loizos Michael。2021年。神经符号集成：一种组合视角。载于 AAAI。5051–5060。[80] Jacopo Urbani、Ceriel Jacobs 和 Markus Krötzsch。2016年。面向大型知识图谱的列式 Datalog 物化。载于 AAAI。258–264。[81] Leslie G. Valiant。1979年。枚举与可靠性问题的复杂性。《SIAM 计算杂志》8，3（1979），410–421。[82] Timothy van Bremen、Anton Dries 和 Jean Christoph Jung。2019年。通过概率逻辑编程对概率数据执行本体中介查询。载于 CIKM。2437–2440。[83] Guy Van den Broeck、Wannes Meert 和 Adnan Darwiche。2014年。用于加权一阶模型计数的斯科伦化。载于 KR。111–120。[84] Maarten Van den Heuvel、Peter Ivanov、Wolfgang Gatterbauer、Floris Geerts 和 Martin Theobald。2019年。通过缩放解离实现概率数据库中的随时近似。载于 SIGMOD。1295–1312。[85] Jonas Vlasselaer、Guy Van den Broeck、Angelika Kimmig、Wannes Meert 和 Luc De Raedt。2015年。通过 Tp 编译实现概率逻辑程序中的随时推理。载于 IJCAI。1852–1858。[86] Jonas Vlasselaer、Guy Van den Broeck、Angelika Kimmig、Wannes Meert 和 Luc De Raedt。2016年。用于概率逻辑程序推理的 TP 编译。《国际近似推理杂志》78（2016），15–32。[87] Sen Wu、Ce Zhang、Feiran Wang 和 Christopher Ré。2015年。使用 DeepDive 增量式构建知识库。CoRR abs/1502.00731（2015）。[88] Wentao Wu、Hongsong Li、Haixun Wang 和 Kenny Q. Zhu。2012年。Probase：一种用于文本理解的概率分类体系。载于 SIGMOD。481–492。[89] Yuke Zhu、Alireza Fathi 和 Li Fei-Fei。2014年。在知识库表示中推理对象的可供性。载于 ECCV。408–424。

算法3 步骤(I)，其中 $\mathrm{P} = (\mathrm{R}, \mathrm{F}, \pi)$，$\mathrm{I} = \{(\alpha_i, \lambda_i)\}$，且 λ_i 是关于 F 中元素的布尔公式

1: I′ := ∅ Δ := ∅ 2:**对每个**规则 r in P与每个实例化 $\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n$ of r，使得每个(α_i, λ_i) 都属于 I，且存在某个 λ_i **执行**3: **添加** $(\alpha, \bigwedge_i^n \lambda_i)$ 到 Δ 4:**对每个** α，使得形如$(\alpha, \cdot)$ 的一对属于 $\Delta \cup \mathrm{I}$ **执行**5: **添加** $(\alpha, \bigvee_{(\alpha,\varphi)} \varphi) \in \Delta$ 到 I′ 6: **返回**I′

增量计算执行图

我们将详细说明增量计算解释图的步骤（算法1第4行和算法2第4行）。正如 Tsamoura 等人所示，解释图可以增量计算。 [77]假设我们已经构建了一个深度为 $k-1$ 的解释图，并且希望添加深度为 k 的新节点，以免遗漏任何规则执行。由于边决定了我们要在哪些节点上实例化规则的前提，因此我们需要确保扩展后的解释图穷尽地添加所有可能的边。对于在解释图上进行推理时，事实可能实例化规则 r 的前提的深度为 $< k$ 的每种节点组合，我们称其为与 r 的 k 兼容节点：

定义6. *[从 [77]]* 中取 n 个节点的组合 $(u_1, \ldots, u_n)$，其中来自 G，若 $k \geq 2$ 为整数，则该组合与 r k-兼容：

- u_i 头部中的谓词是 p_i；
- 每个 u_i 的深度小于 k；与 • 至少一个节点位于$(u_1, \ldots, u_n)$的深度为 $k-1$。

Th上述思想总结为一个迭代过程，在每一步构建一个深度为 k 的 G^k 解释图。 k:

- **(基础步骤)**如果 $k = 1$，则对于每个基础[3]规则 r，向 G^k添加一个与 r关联的节点 v。• **(归纳步骤)** 否则，对于每个非基础规则 r与来自 G^{k-1}的每个节点组合（u_1, ..., u_n），如果该组合与 r兼容，则向 G^k添加：(i)一个与 r关联的新节点 v，以及 (ii) 一条边 $u_i \rightarrow_i v$，对于每个 $1 \leq i \leq n$。

B 第 4 节的证明

如第4.2节所述，引理 1 涉及算法1和 TcP 的简化版本，其中不使用终止条件。对于算法1，这种简化意味着跳过第9行中的检查，因此每棵树 τ 都会被添加到 tset(v, F) 中。我们在算法3中概述了简化版 TcP 每次迭代期间执行的步骤，其输入为一个实例 I。在全文中，我们固定一个概率逻辑程序 $\mathrm{P} = (\mathrm{R}, \mathrm{F}, \pi)$，并用 G^i 表示算法1第 i 次迭代结束时计算出的解释图，用 $G^i(\mathrm{F})$ 表示存储在 G^i 节点中的推导树。我们还用 I^i 表示 TcP 第 i 次迭代结束时计算出的实例，其中 $\mathrm{I}^0 = \{(f, f) \mid p :: f \in \mathrm{F}\}$。引理 1 的证明依赖于断言1，而该断言同样涉及简化版算法1。

断言1. 设 τ_1、.....、τ_n 是 $G^i(\mathcal{F})$ 中的推导树，其中 $i \geq 0$，且每棵 τ_j 的根为 α_j，并存储在深度为 $< i$ 的节点中。再设 r 是一条规则，$\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n$ 是该规则的一个实例化。那么，对每个 $i \geq 0$，以下归纳假设成立：

- ρ。存在一棵树 τ，属于 $G^i(\mathcal{F})$，使得 τ 的根为 α，且存在一条边，从 τ_j 的根连到 τ 的根。

证明。对于 $i = 0$，归纳假设 ρ 显然成立，因为根据定义，G^0 是空图。对于 $i+1$，假设 ρ 对 $i \geq 0$ 成立，则有如下结论。设 $(u_1, \ldots, u_n)$ 是 G^i 中的节点元组，使得 $\tau_j \in$ tset(u_j, F)，其中 $1 \leq j \leq n$。若每个 u_j 的深度均为 $< i$，则 ρ 显然成立。因此，我们考虑至少有一个 u_j 的深度为 i 的情况。

由于每个τ_j的根是α_j，且由于实例化 $\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n$，根据定义6可知，$(u_1, \ldots, u_n)$与 r i兼容。由于$(u_1, \ldots, u_n)$与 r i兼容，根据算法1第4行的步骤可知，G^{i+1} 包含一个节点 v，因此对于 $1 \leq j \leq n$，边$u_j \rightarrow_j v$属于 G^{i+1}。此外，由于 $\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n$，根据定义2，且由于对于 $1 \leq j \leq n$，τ_j 属于 tset(u_j, F)，$\mathrm{T}(\alpha, v, \mathrm{F})$包含一棵树 τ，其根原子为 α，且边从每个τ_j的根指向 τ的根，其中 $1 \leq j \leq n$。树 τ将被存储在 tset(v, F) 中。由于 $\tau \in$ 属于 tset(v, F)，可知对于 $i+1$，ρ成立，由此完成断言1的证明。□

引理 1. 对于每个 $i \geq 0$，$(\alpha, \lambda_\alpha^i) \in \mathcal{I}^i$ 当且仅当 $\bigvee_{j=1}^m \phi(\tau_j) \equiv \lambda_\alpha^i$，其中 τ_1、.....、τ_m 都是 $G^i(\mathcal{F})$ 中的树，且事实 α 为根。

证明。对于 $i = 0$，归纳假设在两个方向上都成立，因为根据定义，$\mathrm{I}^0 = \{(f, f) \mid p :: f \in \mathrm{F}\}$ 和 $G^0(\mathrm{F}) = \mathrm{F}$，而 F 中一个事实的谱系就是该事实本身。

(⇒)对于 $i+1$，并假设 ω对 i成立，证明过程如下。假设规则 r在 $\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n$下于 TcP的第 $i+1$步实例化，并令 $(\alpha_j, \lambda_{\alpha_j}^i) \in \mathrm{I}^i$，其中 $1 \leq j \leq n$。由于归纳假设对 i成立，我们知道，对于每个 α_j，存在一组树 Tj，它们属于 $G^i(\mathrm{F})$，且具有相同的根 α_j，使得 $\bigvee_{\tau \in \mathcal{T}_j} \phi(\tau) \equiv \lambda_{\alpha_j}^i$成立。此外，根据归纳假设，T$j$中的每棵树都存储在深度为 $\leq i$的节点中，

[3]基础规则的前提中仅引用数据库关系；非基础规则则仅引用派生关系。每个规则集都可以改写为这种形式 [77]。

$1 \le j \le n$。由此可知，在算法3第3行的步骤中，根据上述规则实例化所计算出的公式为

$$\bigwedge_{j=1}^{n} \lambda^{i}_{\alpha_j} \equiv \bigwedge_{j=1}^{n} \bigvee_{\tau \in \mathcal{T}_j} \phi(\tau) \equiv \bigvee_{(\tau_1,\ldots,\tau_n) \in \mathcal{T}_1 \times \cdots \times \mathcal{T}_n} \phi(\tau_1) \wedge \cdots \wedge \phi(\tau_n) \tag{3}$$

根据断言1，我们知道，对于每个元组 $(\tau_1,\ldots,\tau_n) \in \mathrm{T}_1 \cdot \cdot \times \ldots \mathrm{T}n$，存在一棵树 τ属于 $G^{i+1}(\mathrm{F})$，使得 τ具有根事实 α，并且从 τ_j的根到 τ的根存在一条边。根据上述内容以及 ϕ的定义，$\phi(\tau) = \phi(\tau_1) \wedge \ldots \phi(\tau_n)$ 成立。因此，α在 $G^{i+1}(\mathrm{F})$ 中的谱系为

$$\bigvee_{(\tau_1,\ldots,\tau_n) \in \mathcal{T}_1 \times \cdots \times \mathcal{T}_n} \phi(\tau_1) \wedge \cdots \wedge \phi(\tau_n) \tag{4}$$

由于上述原因，ω对 $i+1$成立：首先，在算法3所示步骤的第（$i+1$）次迭代结束时，与原子 α 相关联的公式 λ^{i+1}_{α} 为

$$\lambda^{i+1}_{\alpha} = \lambda^{i}_{\alpha} \vee \bigvee_{\alpha \leftarrow \alpha_1 \wedge \cdots \wedge \alpha_n, s.t. (\alpha_j, \lambda^{i}_{\alpha_j}) \in \mathcal{I}^i} \bigwedge_{j=1}^{n} \lambda^{i}_{\alpha_j} \tag{5}$$

其次，由于 ω，对于所有树 $\tau_1,\ldots,\tau_m$在 $G^i(\mathrm{F})$ 中且根原子为 α 的树，我们有

$$\bigvee_{j=1}^{m} \phi(\tau_j) \equiv \lambda^{i}_{\alpha} \tag{6}$$

最后，$G^{i+1}(\mathrm{F})$ 中 α 的谱系是以下析取式：对于每个带有根事实 α 的 $G^{i+1}(\mathrm{F})$ 中的 τ，其公式 $\phi(\tau)$。（⇐）证明与另一个方向类似，并依赖于断言1和（3）。 □

$\mathrm{P_R}$命题 1。 对于 α、τ 和 $\tau\prime$ 的两棵推导树，如果 $\tau\prime$是 τ 的子树，那么 $\phi(\tau) \vee \phi(\tau\prime) \equiv \phi(\tau\prime)$成立。*ds.*

证明。如果 $\tau\prime$是 τ 的子树，则 $\phi(\tau\prime)$ 是 $\phi(\tau)$ 的子合取项。因此，$\phi(\tau) \vee \phi(\tau\prime) \equiv \phi(\tau\prime)$ 成立。 □

引理 2. PReason($\mathcal{P}$) 对于具有有限赫布兰基底的每个概率程序 $\mathcal{P}$ 都终止。

证明。当一个（概率）逻辑程序具有有限的赫布兰基底时，那么不同的规则实例化的数量是有限的。根据上述内容，如果我们将规则实例化组织成一个图 Γ，使得 Γ包含一条从原子 $\alpha_1,\ldots,\alpha_n$到原子 α的边，其中每个规则实例化为 $\alpha \leftarrow \alpha_1,\ldots,\alpha_n$，则会存在一个深度 $k > 0$，使得要么不存在规则实例化，其中规则前提中的原子的深度为 $< k$；要么存在这样的实例化，但原子推导中出现了重复，即深度为 α 的推导原子 k 在其祖先节点集合中包含 α。上述内容表明，对于具有有限赫布兰基底的程序，算法1确实会终止。注意，（概率）Datalog程序始终具有有限的赫布兰基底，因此算法1始终会终止。□

定理 2。 PReason($\mathcal{P}$) 是任意概率程序 $\mathcal{P}$的一个谱系 TG。

证明。回顾定义 3，对于概率程序 P，EG G是 P 的一个谱系 TG，当且仅当对于每个原子 $\alpha \in \mathrm{HB(P)} \setminus \mathrm{F}$，$G(\mathrm{F})$ 中 α的谱系与 P 中 α的谱系逻辑等价。引理 1 保证，在每一步$\mathcal{P}$ 中，基于推导树计算出的公式与 P 的最小参数化模型中的 λ 公式的析取项等价（假设两种方法都不进行终止检查）。此外，引理 2 表明算法1 会终止，因此根据引理 1，它与 P 的最小参数化模型等价。□

推论 3。 对于每个概率程序 $\mathcal{P}$ 和每个原子 $\alpha \in HB(\mathcal{P})$，其在 $G^k(\mathcal{F})$ 中的谱系的概率小于其在 $\mathcal{P}$中的概率 α。

证明。这直接得出。 进一步地，基于以下事实：原子 α在 $G^k(\mathrm{F})$ 中的谱系是其在 $G(\mathrm{F})$ 中完整谱系的一个析取项。 □

C 第 5 节的证明

定理 4。 对于每个概率程序 $\mathcal{P}$，PCOReason($\mathcal{P}$) 都是 $\mathcal{P}$ 的谱系 TG。

证明。定理 4 的证明源于引理 3、4 和 5，我们将在下文予以证明。在全文中，我们固定一个概率逻辑程序$\mathrm{P} = (\mathrm{R}, \mathrm{F}\ \pi)$，并将 G^i_1 与 G^i_2记为在算法1和算法2的第 i 次迭代结束时计算得到的执行图。

引理 3. 对于每个 $i \ge 0$，以下归纳假设成立：

对于每个推导树 τ，其根事实 α 存储在深度为 i 的节点中 G^i_1，都存在一个推导树 ε，该推导树存储在深度为 i 的节点中 G^i_2，使得 $\tau \in$ 展开(ε)。

证明。对于 $i=0,\xi_1$，显然成立，因为 G_1^0和 G_2^0都为空。对于 $i+1$，并假设 ξ_1对 $i\geq 0$成立，证明如下。设 τ是在算法1的第 $i+1$ 次迭代结束时添加到节点 v 的推导树。假设 v与规则 r 相关联，且 τ 的根是事实 α。根据第5节中冗余的定义以及算法1第9行和第10行的步骤，可知 τ 相对于 α 并不冗余。此外，根据定义2，可知存在 r 的一个实例化 $\alpha\leftarrow\alpha_1\wedge\ldots\wedge\alpha_n$，使得对于每个 $1\leq j\leq n$，(i) 存在一个深度为 $\leq i$ 的节点 u_j，(ii) 一棵树 τ_j 存储在 tset(u_j, F) 中，且 (iii) 边 $u_j\rightarrow v$ 位于 G_1^{i+1} 中。由于算法1第4行的步骤，$u_1,\ldots,u_n$ 中至少有一个节点的深度为 i。由于 ξ_1 对 $\leq i$ 成立，因此对于每个 τ_j，存在一棵推导树 ε_j，存储在 G_2^i 中的节点 $u'j$ 里，其深度与 u_j 相同，且 $\tau_j\in$ 展开(ε_j)。由于算法2第4行的步骤，且因为 $u_1,\ldots,u_n$ 中至少有一个节点的深度为 i，并且来自 G_1^i 的节点组合 $(u_1,\ldots,u_n)$ 与 r 是 $(i+1)$-兼容的（见定义6），所以来自 G_2^i 的节点组合 $(u'1,\ldots,u'n)$ 与 r 是 $(i+1)$- 兼容的。因此，在算法 2 第 $(i+1)$ 次迭代开始时，$(u'1,\ldots,u'n)$ 将在第 4 行中被考虑。由此，G_2^{i+1} 将包含一个与规则 r 相关联的节点 v'，以及一条从 $u'j\rightarrow v'$ 出发的边，其中 $1\leq j\leq n$。此外，由于对于每个 $1\leq j\leq n$，ε_j 存储在 $u'j$ 中，使得 $\tau_j\in$ 展开(ε_j)；由于 ε_j 的根是 α_i；由于 $\alpha\leftarrow\alpha_1\wedge\ldots\wedge\alpha_n$ 是 r 的一个实例化；并且根据定义2，可知对于每个 $1\leq j\leq n$，T(α,v', F) 将包含一棵树 ε，而 G_2^{i+1} 将包含一条边 $u'j\rightarrow v'$。我们区分以下几种情况。

- 不进行折叠。因此，在算法2的第10行中，Z =T(α，v'，F)。然后，算法2将在第11行 ε进行迭代。根据定义5，我们知道，对于每个 (δ_1，……，δ_n) ∈展开(ε_1) ×⋯ ×展开(ε_n)，展开(ε)都包含一棵树 δ，使得根(δ) 是 α，且从每个根(δ_i)到根(δ)都有一条边。鉴于上述情况，且由于 τ_j位于展开(ε_j)中，对于 $1\leq j\leq n$，可知 τ位于展开(ε)中。由于 $\tau\in$展开(ε)，且 τ相对于 α 并非冗余，并根据第5节中冗余的定义，ε相对于 α并非冗余，因此，ε会在算法2的第13行被添加到 tset(v'，F)中。
- 折叠。根据前一种情况的每其他树折叠为一棵新树 ε'。因此，ε' 位于 unfold(ε') 中，并且 ε 相对于 α 并非冗余。由于 unfold(ε') ⊇unfold(ε) ⊇{τ}，τ 将包含在 ε' 的展开中。由于 $\tau\in$unfold(ε')，并且 τ 相对于 α 并非冗余，因此 ε' 相对于 α 并非冗余。因此，在算法2的第13行，ε' 被添加到 tset(v',F) 中。

T上述内容表明，ξ_1 对 $i+1$ 成立，从而完成引理 3 的证明。□

引理 4. 对于每个 $i\geq 0$，以下归纳假设成立：

- ξ_2。对于每棵根事实为 α 的推导树 ε，若其存储在 G_2^i 中深度为 i 的节点上，则 展开(ε) 中每棵相对于 α 并非冗余的树，都存储在 G_1^i中深度为 i 的节点上。

证明。与归纳假设 ξ_1的证明类似，ξ_2 对 $i=0$ 显然成立，因为 G_1^0 和 G_2^0 都为空。对于 $i+1$，并假设 ξ_2 对 i成立，证明如下。设 ε 是一棵推导树，它在算法2的第（$i+1$）次迭代结束时已添加到节点 v。假设节点 v 与规则 r 相关联，且 ε 的根是事实 α。由于 ε 被添加到节点 v，因此 ϵ 相对于 α 并非冗余。此外，根据定义2，存在 r 的一个实例化 $\alpha\leftarrow\alpha_1\wedge\ldots\wedge\alpha_n$，使得对于每个 α_j，都有一个 ϵ_j 存储在深度为 $\leq i$ 的某个节点 u_j 中，且 $u_j\rightarrow v$ 属于 G_2^{i+1}。由于算法2第4行的步骤，u_1、……、u_n 中至少有一个节点的深度为 i。由于 ξ_2 对 $\leq i$ 成立，对于每个 ε_j，其中 $1\leq j\leq n$，展开(ε_j) 中每棵相对于 α 非冗余的树，都存储在 G_1^i 中的一个节点 $u'j$ 处。此外，由于 ξ_2，$u'j$ 与 u_j 的深度相同。

要证明 ξ_2对 $i+1$成立，我们需要证明，unfold(ε) 中相对于 α不冗余的每棵树都存储在 G_1^{i+1}中深度为 i的某个节点中。根据定义5，unfold(ε) 对每个 $(\delta_1,\ldots,\delta_n)$ ∈unfold(ε_1) ×⋯ ×unfold(ε_n) 都包含一棵树 δ，其根为 α，并且从每个 root(δ_i) 到 root(δ) 都有一条边。由上述内容，根据定义2，根据对 r的实例化 $\alpha\leftarrow\alpha_1\wedge\ldots\wedge\alpha_n$，并且由于每个 u_j与 $u'j$具有相同的深度，对于 $1\leq j\leq n$，可知节点组合 $(u'1,\ldots,u'n)$ 都与 r兼容。由上述内容以及算法1第4行的步骤可知，对于每个组合 $(\tau'1,\ldots,\tau'n)$，其中 $\tau'j\in$unfold(ε_j)，对于 $1\leq j\leq n$，G_1^{i+1}中将存在一个深度为 $i+1$的节点 v'和一棵树 τ'，使得 $\tau'\in$T(α,v', F)，并且从 τ'的根到每个 $\tau'j$的根都有一条边。最后，T(α,v', F) 中相对于 α不冗余的每棵树，都将在算法1第10行存储于 tset(v', F) 中，从而证明 ξ_2对 $i+1$成立。□

引理 5. 对于概率程序 $\mathcal{P}=(\mathcal{R}、\mathcal{F}\ \pi)$，PReason($\mathcal{P}$) 在步骤 i 当且仅当 PCOReason($\mathcal{P}$) 在步骤 i终止。

证明。(↛（反证法）假设 PReason(P) 在步骤 i 终止。那么，对于每个节点 v，算法1第 8 行中计算的所有推导树相对于 α 都是冗余的。因此，对于每个 $v$in G_1^i，tset(v，F) = ∅。为达到矛盾，假设 PCOReason(P) 未在步骤 i 终止。这意味着算法2第 13 行中的某个 ε 相对于 α 不是冗余的。根据上述内容，在 G_1^i(F) 中至少存在一棵树 τ'∈unfold(ε)，其相对于 α 不是冗余的。然而，如果存在这样的 τ'，那么根据引理 4，算法1应当在第 8 行遍历它，并最终在第 10 行将其添加到相应的节点中，从而产生矛盾。

（⇐）（反证法）假设 PCOReason(P) 在步骤 i 终止。于是，对于 G_2^i 中的每个节点 v，都有 tset(v，F) = ∅。反设 PReason(P) 未在步骤 i 终止。那么，在算法1第 8 行访问的某棵树 τ 相对于 α 并非冗余。然而，根据引理 3，我们知道树 τ 将出现在 G_2^i(F) 中某棵推导树 ε' 的展开中。此外，由于 τ 相对于 α 并非冗余，因此可知 ε' 相对于 α 也并非冗余。因此，应将 ε' 存储在相应节点中（见算法2第 13 行），从而得出矛盾。□

在证明引理 3、4 和 5 之后，我们可以回到定理 4 的证明。根据引理 5，我们知道算法1和算法2在同一次迭代中终止。令该次迭代为 i。对于一个原子 α，令 τ_1、……、τ_n 为 $G_1^i(\mathrm{F})$ 中根为 α 的所有树。根据引理 3，对于每个 τ_j 和 $1 \le j \le n$，存在一棵推导树 $\varepsilon_j \in G_2^i(\mathrm{F})$，使得 $\tau_j \in$展开(ε_j)。此外，根据引理 4，每棵 $\delta \in$展开(ε_j) 的树要么是 τ_1、……、τ_n 中的一棵，要么相对于 α 是冗余的。令 $\tau\prime 1$ 、 …… 、 $\tau\prime m$ 为 $\bigcup_{j=1}^{n}$ 展开 (ε_j) 中相对于 α 冗余的所有树。根据上述内容，我们有 α 在 $G_2^i(\mathrm{F})$ 中的谱系为：

$$\bigvee_{j=1}^{n} \bigvee_{\delta \in \mathsf{unfold}(\varepsilon_j)} \phi(\delta) = \bigvee_{j=1}^{n} \phi(\tau_j) \vee \bigvee_{j=1}^{m} \phi(\tau'_j) \tag{7}$$

由于算法1的正确性，对于每棵这样的树 $\tau\kappa$，对于 $1 \le \kappa \le m$，都存在一棵树 τ_ℓ，其中 $1 \le \ell \le n$，使得 τ_ℓ是 $\tau\kappa$ 的子树——否则， α 在 $G_1^i(\mathrm{F})$ 中的谱系就不会不完整。由命题1及上述内容可得

$$\bigvee_{j=1}^{n} \phi(\tau_j) \vee \bigvee_{j=1}^{m} \phi(\tau'_j) \equiv \bigvee_{j=1}^{n} \phi(\tau_j) \tag{8}$$

公式（8）表明，原子 α 在 $G_1^i(\mathrm{F})$ 中的谱系与 α 在 $G_2^i(\mathrm{F})$ 中的谱系逻辑等价。由上述内容可知，并且由于对于每个原子 $\alpha \in \mathrm{HB(P)} \setminus \mathrm{F}$， α 在 $G_1^i(\mathrm{F})$ 中的谱系与 α 在 P 中的谱系逻辑等价（见定理 2），因此， α 在 $G_2^i(\mathrm{F})$ 中的谱系将与 α 在 P 中的谱系逻辑等价。因此，算法2也将为概率程序 P 计算出一个谱系 TG，从而完成定理 4 的证明。□

D 附加实验结果

查询生成。 我们应用了一种名为 QueryGen 的技术，用于生成复杂度不断增加的合成查询。该技术与 [50]所应用的技术非常相似。它也与 [10], 中使用的技术相似，并在此基础上对回答合成查询时所涉及的推理程度进行了更好的控制。

QueryGen 接收一组规则 R 和一组事实 F 作为输入，并返回一组针对派生关系的查询。该技术首先计算图 O。图 O 为派生关系中出现的每一列包含一个节点，并在数据存在重叠的每对列之间连一条无向边。由于 O 对派生关系中的重叠列进行编码，其构造基于非概率程序 (R, F) 的模型 M，也就是说，QueryGen 首先计算 (R, F) 的模型，然后找出派生数据中的重叠部分。

计算出 O后，QueryGen 在 O 上执行随机游走，以计算初始查询集 Q，其中每个查询涉及至多 P 个派生谓词和至多 E 个随机选择的自由变量。在第二步中，该技术针对Q中的每个查询计算：（i）其递归谓词的数量；（ii）定义其每个谓词的规则数量；以及（iii）其任意谓词与外延谓词之间的最大距离。这里所说的谓词，是指每个查询的主体中出现的谓词。标准（i）和（iii）基于R 的依赖图 Δ 计算，其中 Δ 为R 中出现的每个谓词包含一个节点；如果 R中存在一条规则，其结论包含一个 h-原子且其前提包含一个 b-原子，则从谓词 b 到谓词 h存在一条边。具体而言，如果谓词 h 在 Δ中的循环路径上出现，则谓词 h 是递归的；谓词 h 与外延谓词 b 之间的距离定义为 b 与 h之间最长路径的长度。为了创建具有挑战性的基准测试，我们根据标准（i）–（iii）剔除 Q中这些标准取值最低的查询。在第三步中，QueryGen 在 M 上执行剩余查询——请记住， M是程序 (R， F) 的模型——并丢弃空查询。在第四步也是最后一步中，QueryGen 随机选择 Q 中每个非空查询的答案集中的一个常量，并使用该常量绑定相应的自由变量。

下面我们详细说明 QueryGen 中的各个步骤。第一步和第二步起到枢纽作用，用于在 R 中的派生谓词上创建非空查询。根据标准 (i)–(iii) 对查询进行排序，旨在选出推理难度最高的查询：标准 (i)–(iii) 的取值越高，回答这些查询所需的推理就越多。第三步确保查询确实非空，第四步则旨在减少每个查询的答案数量。我们将查询限制为包含 {1、. . .、4} 个原子，且最多包含三个自由变量。

AnyBurl 场景。下面进一步介绍如何根据 AnyBurl 规则创建概率数据库。AnyBurl 是一种规则挖掘技术，因此它会为挖掘出的规则而非知识库事实标注置信度。在我们的设定中，与概率相关联的是事实，而非规则。不过，这并不构成问题。正如第 2 节所述，有一种简单的方法可以将这类规则集转换为概率程序。具体做法是在规则的前提中添加一个额外的“虚拟”事实，并将其概率设为该规则的置信度。根据训练三元组和验证三元组创建的知识库事实，其概率均设为 1。

Scallop. 在 LUBM 场景中，我们使用了 Scallop 的 GitHub 发布版本[4]，因为 NeurIPS 网站[5] 上的版本不提供用于指定规则和事实的接口。我们没有在 DBpedia 和 Claros 中使用 Scallop，因为 GitHub 发布版本不支持如此大型的程序，在加载数据时会抛出异常。作者将 Scallop 扩展到超大数据规模列为未来工作 [49]。此外，对于 Smokers 场景，Scallop 可能会遗漏推理深度在 k 阈值以内的一些证明。我们已将上述所有问题告知作者。对于 VQAR 场景，我们使用了 NeurIPS 网站上的引擎，因为 GitHub 上的版本不支持三元谓词。

[4]https://github.com/scallop-lang/scallop-v1. [5]
https://proceedings.neurips.cc/paper/2021/hash/d367eef13f90793bd8121e2f675f0dc2-Abstract.html.

表 7：与基准测试查询相关的最小和最大推理深度（DP）、推导数（DR）和规则数（R）。统计数据仅涵盖未超时或未出现内存不足的查询。对于 VQAR，DR 列显示的是合并谱系后的推导数。

	LUBM010	LUBM100	DBpedia	Claros	YAGO5	YAGO10	YAGO15	WN18RR5	WN18RR10	WN18RR15	Smokers4	Smokers5	VQAR
最小/最大动态规划	3/22	3/22	4/19	5/25	6/12	6/12	6/14	8/12	7/12	4/14	4	5	15/15
最小/最大动态推理	17/107M	17/117M	26/691k	64/502k	5/91k	8/48k	19/116k	17/727	12/105	12/2k	74/364	112/1138	370k/390k
最小/最大 R	2/208	2/208	611/5552	6613/6672	35/275	86/1217	95/2104	147/216	257/440	95/720	3/3	3/3	6/6

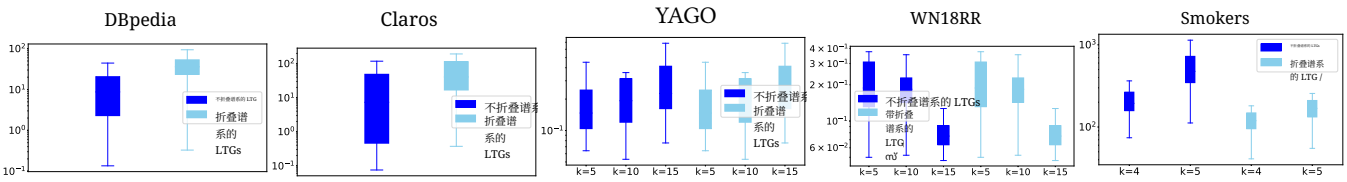


图 8：使用 LTG 在不同场景下进行谱系收集所需的时间。