利用 d-DNNF 对特征模型进行重复计数查询

Chico Sundermann[1], Heiko Raab[1], Tobias Heß[1], Thomas Thüm[1], 与 Ina Schaefer[2]

[1]乌尔姆大学，德国[2]卡尔斯鲁厄理工学院，
德国

2023年3月23日

arXiv:2303.12383v1 [cs.SE] 22 Mar 2023

摘要

特征模型常用于描述产品线的有效配置。在工业领域，由于特征和约束数量众多，特征模型往往十分复杂。因此，人们提出了大量自动化分析方法。其中许多方法依赖于计算有效配置的数量，这通常需要求解 #SAT 问题，而这是一项计算开销很大的操作。此外，大多数基于计数的分析都需要对同一特征模型进行大量 #SAT计算。具体而言，许多分析需要通过多次计算，评估包含特定特征或符合部分配置的有效配置数量。与其针对高度相似的公式反复进行开销巨大的计算，我们致力于通过在这些计算之间重用已有知识来提升性能。在这项工作中，我们首次提出重用 d-DNNF，以高效地对特征和部分配置执行重复查询。实证评估表明，与反复调用 #SAT 求解器这一当前最佳水平相比，我们的方法速度最高可提升 8,300 倍（节省 99.99% 的 CPU 时间）。与使用 #SAT 求解器相比，应用我们的工具 ddnnife 可将运行时间从数天缩短至数分钟。

1 引言

产品线描述一组相关产品，这些产品共享一组特征 [1–5]。通常，并非这些特征的每种配置都是有效的（即代表实际产品）。特征模型是规定有效软件配置集合的事实标准 [1, 4, 6]，也常用于其他领域 [7–10]。一般而言，特征模型由基础特征以及描述特征间依赖关系的一组约束构成。由于特征和约束数量庞大，通常无法人工分析特征模型 [1]。因此，人们需要自动化分析，例如检查给定配置是否有效 [1, 11–16]。

许多与实践相关的分析都依赖于计算某个特征模型的有效配置数量 [17–22]。为了使用标准化求解器，特征模型通常会被转换为命题逻辑进行分析 [23, 24]。SAT求解器用于检查可满足性，而 #SAT 求解器则专门用于计算满足赋值的数量 [20,25]。尽管 #SAT 在计算上十分复杂 [26–29], ，但在近期的研究中 [25]，我们发现，最先进水平的 #SAT 求解器能够在几秒内计算出许多真实世界特征模型的有效配置数量。

我们在先前工作中 [17] 发现，大多数应用都需要进行大量 #SAT调用。例如，在资源分配中，优先考虑出现在许多有效

配置中的特征可能会带来益处。若将现成的 #SAT 求解器作为黑盒使用，则可能需要针对每个特征 [17, 18]分别执行一次 #SAT 计算。文献中已探讨过多种用于常规 SAT 的增量求解器（即在查询之间复用信息） [30, 31], ，但尚未研究增量 #SAT。已有报告指出，特征模型的规模可达数万个特征 [10, 25], ；若重复计数查询不复用信息，可能需要数天的计算时间 [25]。虽然可以在一定程度上缓解如此长的运行时间（例如通过并行化），但特征模型分析常用于交互式应用 [7, 12, 14–16] 或持续集成环境 [32,33], ，因此必须大幅缩短运行时间并限制资源使用。尤其是，这些需求源于我们与汽车行业的合作；在合作中，我们集成了原型，以降低资源消耗。

随着特征模型不断增大 [25, 34, 35], ，人们因此希望缩短分析所需的运行时间。我们提出采用知识编译，以降低执行多个查询时所需的计算开销 [36–39]。具体而言，原始公式会在离线阶段被转换为目标格式。之后，便可使用生成的格式进行在线计算（例如 #SAT），其复杂度通常相对于知识编译产物的大小呈多项式时间 [36]。

确定性可分解否定范式（d-DNNF）能够在线性时间内计算满足赋值的数量，所需时间与其节点数成正比 [40]。使用现有工具将公式编译为 d-DNNF 的复杂度与求解 #SAT 问题相近，因为 d-DNNF 是根据一次 #SAT 调用的执行轨迹构造的 [37–39]。在我们近期的工作中 [25],，我们表明，对于特征模型，d-DNNF 编译器 [37–39, 41, 42]通常比其他支持模型计数的知识编译器更快。例如，编译为 d-DNNF 的速度显著快于编译为二叉决策图 [43, 44]。因此，d-DNNF 很适合用于特征模型计数。

尽管编译为 d-DNNF 已得到充分研究 [37–39, 41, 42], ，但对给定 d-DNNF 的复用却鲜有探索 [45, 46]。特别是，此前无论在软件工程还是其他领域，都未评估过将 d-DNNF 用于多个计数查询的复用情况。在相关工作中，Sharma 等人 [46] 遍历 d-DNNF，以推导用于均匀随机采样的满足赋值。此外，一些研究将 d-DNNF 编译器用于计数，把它们当作黑盒 #SAT 求解器，而不复用 d-DNNF；这实际上意味着每个查询都要编译一个新的 d-DNNF [20, 25]。

在这项工作中，我们提出对重复计数查询进行复用d-DNNF，并将其用于计算符合以下条件的有效配置数量：(1) 包含某个特定特征，或 (2) 符合某个部分配置（即包含和/或排除若干特征）。我们通过大规模实证评估，展示了我们的方法相较于最先进水平的巨大改进。具体而言，我们使用现有编译器 [37–39] ，一次性将给定特征模型编译为 d-DNNF，然后复用该 d-DNNF 执行大量基于计数的计算。因此，我们的工作聚焦于高效复用 d-DNNF，并作出以下贡献。

首先，我们介绍针对给定 d-DNNF 的重复计数查询的算法和若干新颖优化（参见第 4.1 节和第 4.2 节）。所介绍算法的一些基础思路源自 Darwiche [45]。不过，Darwiche 的想法此前尚未进一步落实为实现或实证评估。此外，我们还提出了一系列新颖优化，大幅减少了遍历 d-DNNF 所需的次数，并加快了单次遍历的速度。

其次，我们提供开源工具 ddnnife[1] ，其中包含所有介绍的算法和优化（参见第 5 节）。我们的 d-DNNF 推理器 ddnnife 可读取 d-DNNF 作为输入，并支持流行编译器 [37–39] 提供的所有格式。给定 d-DNNF 后，用户即可使用 ddnnife 对其执行计数查询。

第三，我们在 62 个工业特征模型上进行了大规模实证评估，将 d-DNNF 的复用与最先进水平的 #SAT 求解器进行比较（参见第 6 节）。在此过程中，我们将

[1] https://github.com/SoftVarE-Group/d-dnnf-reasoner/releases/tag/March2023

我们的工具 ddnnife 与每个可用的 d-DNNF 编译器相结合，与最先进的解进行比较。所需的数据、工具和框架均已公开，以便他人复现和重复实验。[2]

2 动机示例

本节通过一个小型示例展示依赖多个计数查询的分析用例。图 1 展示了一个简化个人电脑的特征模型。每台个人电脑都需要至少一种文件系统（表示为或组），并且需要且仅需要一种操作系统（表示为可选组）。此外，使用 APFS 或 NTFS 分别需要使用 Mac 或 Windows，这由树下方的跨树约束表示。在此模型中，Linux 可与 FAT32 文件系统一起在一种配置中被选中。Windows 可与 FAT32、NTFS 或两者同时组合。类似地，Mac 出现在三种有效配置中，其中包括 FAT32 和 APFS 的组合。总共有七种有效配置。

计算有效配置的数量可以支持多种分析。在我们最近的工作 [17], 中，我们介绍了依赖于计算有效配置数量的应用，这些应用（1）汇集自文献 [18, 20, 22, 47–49] ，以及（2）受到产业合作的启发。在那项近期工作中，我们确定了 21 种应用，它们与检测错误和异味、采样、可维护性、支持开发、经济估算以及用户指导相关。其中 21 种应用里有 14 种依赖于对同一特征模型进行多次 #SAT 计算。

例如，包含特定特征的有效配置数量可用于量化某个错误的影响 [17, 20]。优先处理出现在更多有效配置中的问题，而不是只出现在部分有效配置中的问题，或许更为合理。例如，FAT32 中的错误会影响五种有效配置，而 NTFS 中的错误只会影响两种有效配置。

特征模型计数也可用于产品线的开发过程 中。例如，优先考虑能够构建更多不同有效配置的特征，或许会带来好处 [17]。请看以下场景。对于图 1 所示的个人电脑产品线，每种操作系统都已实现，开发人员必须决定先实现哪种文件系统。例如，实现 FAT32 后，我们可以构建每一种包含 FAT32 且不包含 NTFS 和 APFS 的有效配置。计算包含和不包含相应特征的有效配置数量后可以看出，实现 FAT32 可产生三种有效配置，而 NTFS 和 APFS 则各只能产生一种。

虽然对于我们的贯穿示例而言，本文展示的两种分析只需少量计数查询，但在实践中，此类分析通常需要对大型特征模型进行大量查询。例如，同质性 [17, 19, 47], 描述的是特征模型所导出的配置之间的相似性，它需要针对每个特征执行一次计数查询。尤其是在处理包含数千个特征的大型工业特征模型时，此类分析可能导致运行时间大幅增加。对于这 14 个需要多次 #SAT调用的应用，重复利用 d-DNNF 执行多次计算可能会带来好处。

3 背景

本节将介绍后续章节所需的背景知识。首先，我们正式介绍特征模型和特征模型计数。其次，我们介绍 d-DNNF 及如何利用其结构进行模型计数。

[2]https://github.com/SoftVarE-Group/exploiting-ddnnfs-eval/releases/tag/2023年3月

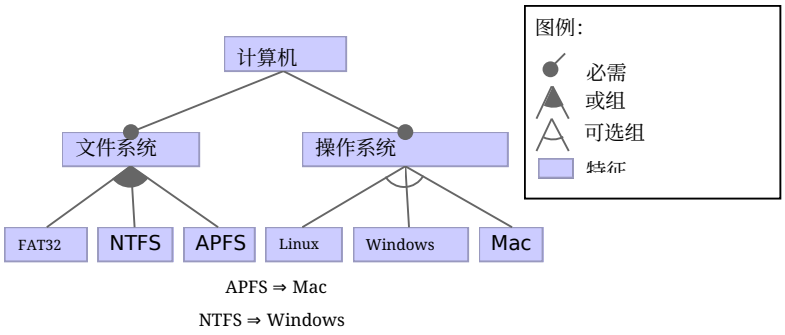

图 1：改编自 Krieter 等人的特征模型 [16]

3.1 特征模型计数

对于给定的产品线，通常使用特征模型来指定有效配置的集合 [1]。形式上，我们将特征模型定义为元组 $FM = (FT, CT)$，其中包含一组特征FT 以及这些特征之间的约束CT。约束集合 CT 包括描述父子关系的层次约束，以及跨树约束。为简单起见，我们将这些约束视为命题公式$propositional$ 的合取（即，只有CT 中的每个约束都得到满足，特征模型才算得到满足）。我们将由此得到的公式称为$F_{FM} = \bigwedge_{c \in CT} c$。众所周知，每种层次约束（即可选、或、必需和可选）都可以转换为命题公式 [1, 23]。

一个配置 $C = (I, E)$ 由两组特征构成，分别描述配置中包含的特征（$I \subseteq FT$）和排除的特征（$E \subseteq FT$）。同一配置中，一个特征不能既被包含又被排除（即 $I \cap E = \emptyset$）。如果所有特征都被包含或排除（即 $I \cup E = FT$），则该配置是完整的；否则，该配置是部分配置 [1]。当且仅当配置 $C = (I, E)$ 是完整的，且满足特征模型 [1]施加的所有约束 CT 时，我们才认为它是有效的。我们将由 FM 诱导的所有有效配置的集合称为 VC_{FM}。

特征模型计数是指计算给定特征模型的有效配置数量。在先前的工作中， [17], 我们介绍了三种特征模型计数类型，用于计算以下有效配置的数量：(1) 整个特征模型的有效配置；(2) 包含某个特征的有效配置；以及 (3) 由部分配置导出的有效配置（即包含某些特征并排除其他特征）。这三种类型涵盖了我们所识别的全部 21 个应用。

定义 1（特征模型的基数）。 特征模型 $FM = (FT, CT)$ 的基数 $\#FM$ 对应于有效配置集合 VC_{FM} 的基数（即 $\#FM = |VC_{FM}|$）。

定义 2（特征的基数）。 特征模型$FM = (FT, CT)$ 中特征 f 的基数 $\#f$ 对应于包含该特征的有效配置集合的基数f（即 $\#f = |\{C = (I, E) \mid C \in VC_{FM}, f \in I\}|$）。

定义 3（部分配置的基数）。 给定特征模型 $FM = (FT, CT)$，部分配置信息 $C_p = (I_p, E_p)$ 的基数 $\#C_p$ 对应于满足以下条件的有效配置集合的基数：包含每个特征 $i \in I_p$ 且排除每个特征 $e \in E_p$（即，$\#C_p = |\{C = (I, E) \mid C \in VC_{FM}, I_p \subseteq I, E_p \subseteq E\}|$）。

许多特征模型分析都可归结为对表示特征模型的命题公式求解 SAT 或 #SAT 问题 [1, 18, 20, 24, 50–52]。例如，特征模型 $FM = (FT, CT)$ 是空的（即不生成任何有效配置 [1]），当且仅当公式 F_{FM}不可满足。上述三种类型的基数都可以归结为调用 #SAT [17, 18]。特征模型的基数 $\#FM$ 等价于满足赋值的数量

算法1 通过 d-DNNF 遍历进行模型计数

```
1: 过程#节点(节点)
2:     如果 节点是 与，则
3:         返回 ∏_{c∈子节点}      #节点(c)
4:     否则如果节点是或则
5:         返回 ∑_{c∈子节点}      #节点(c)
6:     否则如果节点是文字则
7:         返回 1
```

F_{FM} 的赋值数量（即 #SAT(F_{FM})）。特征 #f 的基数可通过 #SAT($F_{FM} \wedge f$) 计算。类似地，部分配置 $C_p = (I, E)$ 的基数 #C_p 可归结为 #SAT($F_{FM} \wedge \bigwedge_{i \in I} i \wedge \bigwedge_{e \in E} \neg e$)。

3.2 使用 d-DNNF 进行模型计数

通常，基于命题逻辑的工具（例如 SAT 和 #SAT 求解器）依赖范式来表示输入公式。最常用的格式是合取范式（CNF） [28, 37–39, 53–61]。然而，CNF 无法在多项式时间内进行模型计数 [26, 36]。

相比之下，*d-DNNF*（确定性 可分解否定 范式 格式 ）支持在与 d-DNNF 节点数量成线性关系的时间内进行模型计数 [36–38]。d-DNNF 语言是否定范式（NNF）的一个子集，其中确定性和可分解性成立 [40]。通常，为了简化和加速针对 d-DNNF 的算法，还会进一步要求 d-DNNF 是平滑的 [36, 38, 45, 55]。

定义4 （否定范式 [40]）。 当且仅当否定范式 中的命题公式 F 仅包含、 $\vee$、 和 $\neg$ 这些逻辑运算符，并且否定（$\neg$）只直接出现在变量前面时，该公式才属于否定范式。

算法1展示了如何计算给定 d-DNNF 的满足赋值数量。从根节点出发，该算法遍历 d-DNNF，并根据节点类型（即合取、析取和文字）应用相应规则，同时利用可分解性、确定性和平滑性。在下文中，我们将解释 d-DNNF 的这三种性质及其对算法1的影响。

定义5 (可分解性 [37])。 命题公式 F 是可分解的 当且仅当对于每个合取 $C = C_1 \wedge C_2 \wedge \ldots \wedge C_n$，每个合取项 C_i 的变量集合彼此不相交（即，$\forall i,\ j\colon i \neq j\colon vars(C_i) \cap vars(C_j) = \emptyset$）。

利用可分解性，合取的满足赋值数量等于其子节点的满足赋值数量之积，如算法1第2至3行所示。如果公式 F 是可分解的，则每个合取中的合取项不共享变量。因此，我们可以将各合取项的满足赋值两两组合（即构建笛卡尔积）。由于这种两两组合，所得集合的大小（即满足赋值的数量）等于相应合取项大小之积。

考虑示例公式 $F = (A \vee B) \wedge (C \vee \neg C)$。左侧部分 $A \vee B$ 产生三个满足赋值 $\{A, B\}$, $\{A, \neg B\}, \{\neg A, B\}$。右侧部分 $C \vee \neg C$ 产生两个满足赋值 $\{C\}, \{\neg C\}$。将两组赋值两两组合后，得到 $3 \cdot 2 = 6$ 个满足赋值，用于 F（即 $\{A, B, C\}$, $\{A, B, \neg C\}$, $\{A, \neg B, C\}$, $\{A, \neg B, \neg C\}$, $\{\neg A, B, C\}$,$\{\neg A, B, \neg C\}$）。如果不具备可分解性，所述过程可能会产生错误结果，因为组合后的赋值可能相互冲突或不互斥，从而导致计数错误。

定义6（确定性 [37]）。当且仅当对于每个析取 $D = D_1 \vee D_2 \vee \ldots \vee D_m$，析取项之间没有共同的满足赋值（即 $\forall i, j\colon\ i \neq j\colon\ D_i \wedge D_j = \bot$）时，命题公式 F 是 确定的 。

利用确定性，析取的满足赋值数量等于其子节点数量之和，如算法1第 4–5 行所示。对于析取，满足其中一个析取项的每个赋值也满足该析取。因此，满足赋值集合是各合取项集合的并集。对于确定性析取，各析取项之间没有共同的解。因此，将所有析取项的满足赋值集合合并时，不会出现重复项。

考虑示例公式 $F = (A \wedge B) \vee (\neg B)$。$(A \wedge B)$ 和 $(\neg B)$ 分别导出 $\{A, B\}$ and$\{A, \neg B\}$, $\{\neg A, \neg B\}$ 作为满足赋值。F 的满足赋值集合是这两个集合的并集 {$\{A, B\}$, $\{A, \neg B\}$, $\{\neg A, \neg B\}$}。根据定义，这些集合彼此不相交，因此满足赋值的数量等于其子节点的满足赋值数量之和。如果没有确定性，所述过程可能会产生错误结果，因为重复的满足赋值可能会错误地增加满足赋值的数量。

定义7（平滑性 [40]）。命题公式 F 是平滑的，当且仅当对每个析取 $D = D_1 \vee D_2 \vee \ldots \vee D_m$，其析取项具有相同的变量集合（$\forall i,\ j\colon\ vars(D_i) = vars(D_j)$）。

为简化使用 d-DNNF [36–38, 45]时的算法，通常会对析取强制要求平滑性。根据定义，对于平滑析取，每个析取项的变量集合都相同。如果不满足平滑性，某个析取项 D_i 中包含一个变量 v，而另一个析取项 D_j中没有出现该变量，那么在 D_j的上下文中，v 可以自由赋值（即取 $\top$ 或 $\bot$）。在这种情况下，D_j 的满足赋值数量需要针对每个缺失变量乘以2。因此，算法1中的第5行需要进行调整，以处理自由变量。如果假设满足平滑性，该算法就无需跟踪变量集合。根据 Darwiche [40], 的结论，每个 d-DNNF 都能在关于其节点数和变量数的多项式时间内转换为平滑形式。除非另有说明，本文均将 d-DNNF 视为平滑的。

每个文字都会引出且仅引出一个满足赋值（即将相应变量设为 $\top$（针对 f）或 $\bot$（针对 $\neg f$）），如算法1第 6–7 行所示。由于 d-DNNF 具有可分解性、确定性和平滑性这些性质，可以使用算法1遍历 d-DNNF，以计算命题公式 [45] 的满足赋值数量。特征模型 FM =（FT CT）的基数 #FM 等同于总体满足赋值数量 #F_{FM}。因此，可以直接使用该算法，通过表示F_{FM} [43, 46] 的 d-DNNF，计算整个特征模型的有效配置数量。在第 4 节中，我们将对该算法进行调整，以计算特征和部分配置的基数。

4 重用 d-DNNF 计算特征和部分配置的基数

在这项工作中，我们利用将知识编译为 d-DNNF 的优势来进行特征模型计数。具体而言，我们关注重复查询，例如特征基数和部分配置所需的查询。对于给定的特征模型，我们使用现有编译器 [37–39]，在一次性的离线阶段将其编译为 d-DNNF。之后，我们通过将两类基数查询转化为对该 d-DNNF 的线性时间查询（相对于 d-DNNF 节点数量），重复利用编译后的 d-DNNF。第 4.1 节中介绍的相应 d-DNNF 查询受到 Darwiche [45] 提出的一种算法思路启发，但此前从未实现或评估过。此外，我们在第 4.2 节中针对这些查询提出了若干优化。

4.1 针对特征和部分配置的 d-DNNF 查询

算法 2 和算法 3 分别展示了通过 d-DNNF 遍历计算特征基数和部分配置基数的过程。这两种算法都对算法1进行了调整，针对文字执行不同的计算。请注意，第 1–6 行在两种算法中完全相同。图 2 展示了一个表示公式 $A \wedge ((B \wedge \neg C) \vee (\neg B \wedge C)) \wedge (D \vee \neg D)$ 的 d-DNNF。每个节点对应一个逻辑运算符或文字。节点旁的三个数字表示该节点在三类基数查询示例中的满足赋值数量。在图 2 中，黑色数字表示遍历 d-DNNF 以计算满足赋值总数（等价地说，即有效配置总数），如算法1所示。图 2 中 d-DNNF 根节点的结果表明，基数为 $\#F_{FM} = 4$。

算法 2 特征基数

```
1: 过程#节点(节点)
2:     如果 节点是 与 则
3:         返回 ∏_{c∈子节点} #节点(c)
4:     否则如果节点是或则
5:         返回 ∑_{c∈子节点} #节点(c)
6:     否则如果节点是文字则
7:         返回#literal_feat(node)
8: 过程#literal_feat(literal, feature)
9:     如果literal.variable ==特征 则
10:        如果literal.正则返回1
11:        否则返回 0
12:    否则返回 1
13:
14:
15:
16:
```

算法 3 部分配置的基数——置

```
1: 过程节点(node)
2:     如果 节点是 与 则
3:         返回 ∏_{c∈子节点} #节点(c)
4:     否则，如果 节点是 或 则
5:         返回 ∑_{c∈子节点} #node(c)
6:     否则，如果节点是文字，则
7:         返回#literal_pc(node)
8: 过程 #literal_pc(literal, in-包含，排除)
9:     如果文字.变量 ∈ 已包含则
10:        如果 文字.为正则返回 1
11:        否则返回 0
12:    否则如果文字.变量 ∈ 已排除则
13:        如果 文字.为正则返回 0
14:        否则返回 1
15:    else
16:        返回 1
```

特征 为了计算特征 $f \in$ 在特征模型 $=$(FT , CT) 中的基数 $\#f$，我们需要排除不包含 f （即 $f \notin I$）的有效配置 $C = (I, E)$，其特征为 $C \in$ VC。因此，在遍历 d-DNNF 以计算 $\#f$ 时，我们通过将负文字 $\neg f$ 设为零，舍弃每个使 f 为假的满足赋值。在图 2 中，蓝色数字表示 B 的基数。文字 $\neg B$ 初始化为零。 B 的正文字以及

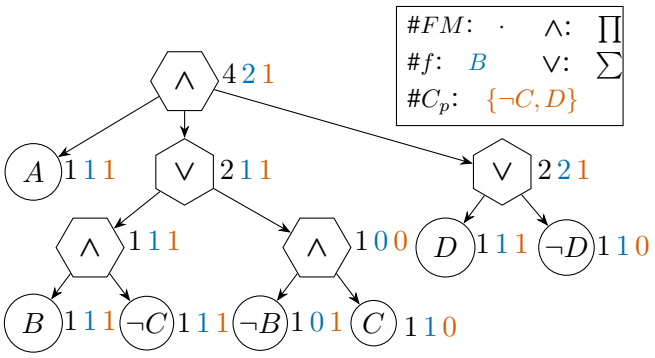

图 2：在 d-DNNF 上计算基数

其他文字均初始化为一。算法 2 第 8–12 行描述了特征基数计算中对文字的处理。使用调整后的文字值，通过遍历 d-DNNF，可以计算包含 B 的满足赋值数量；如图 2 的根节点所示，其基数为 $\#f = 2$ 。

部分配置 对于基数 $\#C_p$ 的部分配置 $C_p = (I_p, E_p)$，我们排除那些将已包含特征赋值为假的赋值 $i \in I_p$ ，或将已排除特征赋值为真的赋值$e \in E_p$。在图 2 中，橙色数字表示部分配置$\{\neg C, D\}$的基数。对于每个已包含的特征（此处： D），正文字和负文字分别初始化为 1 和 0。对于每个已排除的特征（此处： C），正文字和负文字分别初始化为 0 和 1。对应于既未包含也未排除的特征的文字始终初始化为 1。算法 3 第 8–16 行规定了部分配置基数所涉及文字的处理方式。初始化文字节点的值后，可以遍历 d-DNNF 来计算基数 $\#C_p = 1$ ，如图 2 的根节点所示。

正确性 为证明我们提出的算法 2 和算法 3 的正确性，我们假定原始算法（算法1）是正确的 [36, 45]。算法 1 的正确性取决于可分解性、确定性和平滑性这三项性质。应用针对特征和部分配置的假设时，我们实质上是对每个感兴趣的变量执行决策传播。换言之，对于一个决策 d ，我们将每个冲突文字 $\neg d$ 替换为 $\bot$ （假），并将文字 d 替换为 $\top$ （真）。因此，修改后的公式表示满足这些假设的满足赋值子集。要证明算法 2 和算法 3 的正确性，还需说明应用假设后，修改后的公式仍然满足：（1）可分解性；（2）确定性；（3）平滑性。首先，由于我们只移除变量，因此可分解性不会遭到破坏。由此可知， $\wedge$节点中每个合取项对应的变量集在此后仍互不相交。其次，每个确定性$\vee$节点仍具有确定性。通过决策传播，我们不会添加新的满足赋值，只会移除它们。因此，应用假设前互不相交的两组满足赋值，应用假设后仍然互不相交。第三，对于平滑的 $\vee$节点，我们会为每个子节点移除相同的变量集。因此，若应用假设前各变量集相等，应用假设后它们仍然相等。总之，使用算法 2 和算法 3 遍历修改后的 d-DNNF 会得到正确结果，因为修改后的公式表示符合查询的满足赋值集合，且仍是平滑的 d-DNNF。

总结 借助所提出的算法，可以重复利用 d-DNNF，根据特征基数和部分配置，为每个应用计算结果。所述每种算法都基于已编译的 d-DNNF 进行在线计算，其运行时间与 d-DNNF 节点数呈线性关系。

4.2 优化

对于第 4 节中介绍的算法，每次查询都需要遍历给定 d-DNNF 的每个节点。尽管每次查询的复杂度与节点数量呈线性关系，但实际分析可能会对 d-DNNF 发起大量查询，而这些 d-DNNF 可能包含数百万个节点。此外，该算法在每一步都需要处理位数多达数千位的大数 [25]。我们提出了若干优化措施，以（1）减少所需的遍历次数，（2）避免对等价节点进行重复计算，以及（3）减少每次查询所遍历的 d-DNNF 节点占比。

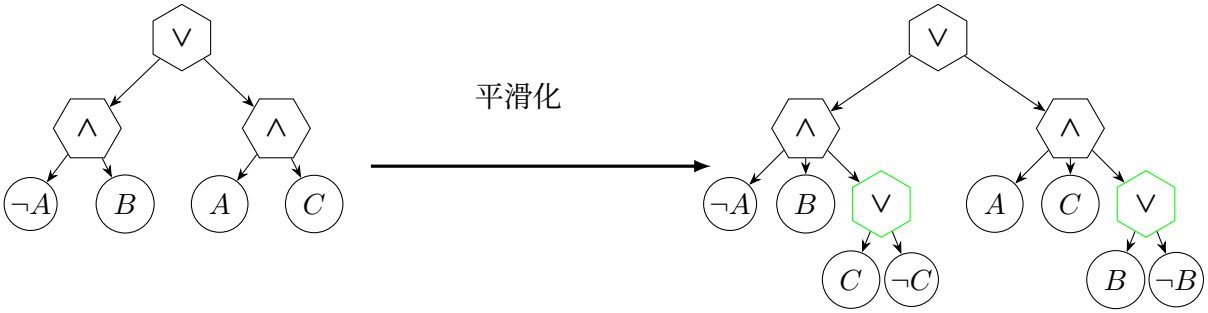

图3：对 d-DNNF 进行平滑化

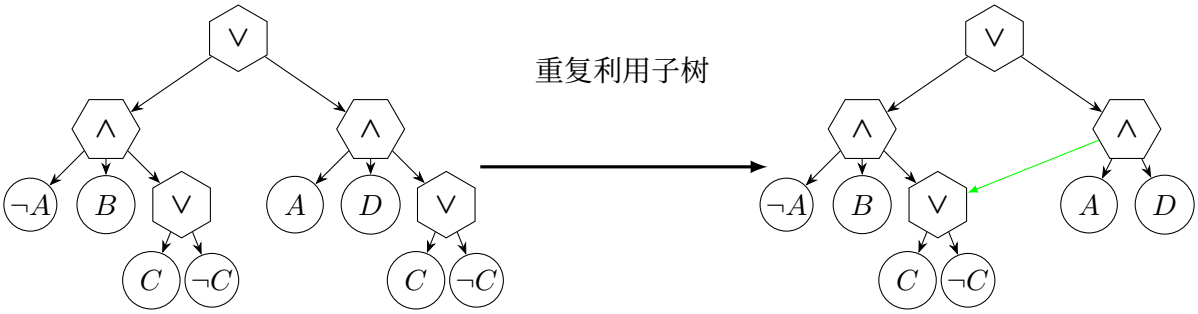

图4：重复利用 d-DNNF 的子树

平滑化 如第3.2节所述，在非平滑 d-DNNF 上计算基数会带来额外的计算开销，因为算法在遍历过程中需要跟踪变量集合 [36]。为确保平滑性，我们在预处理步骤中，为每个非平滑 $\vee$节点添加专用节点，如图 3 所示。具体而言，对于 $\vee$节点的某个子树 Φ 中缺失的每个变量 $v \in missingVariables_{\Phi}$，我们都会添加一个析取 $v \vee \neg v$，从而得到右侧所示的平滑析取。

$$\bigvee_{\Phi \in children} \Phi \quad \rightarrow \quad \bigvee_{\Phi \in children} (\Phi \wedge \bigwedge_{v \in missingVariables_{\Phi}} v \vee \neg v)$$

添加的子公式都是重言式，因为对于每个变量 v，$v \vee \neg v \equiv \top$ 都成立。因此，向子公式 Φ 添加平滑节点时，结果 $\Phi \wedge \top$ 等价于 Φ。在图 3 的示例中，左侧子树不包含变量 C。同理，右侧子树不包含 B。因此，子树的变量集合并不相同，这违反了平滑性。于是，我们分别将平滑节点 $C \vee \neg C$ 和 $B \vee \neg B$ 与这些子树合取。经过平滑化步骤后，根节点 $\vee$ 的两侧 $\{A, B, C\}$ 都包含相同的变量集合，同时公式的语义保持不变。

重复利用子树 实际上，d-DNNF [37–39] 中会多次包含相同的子树。如果在计算查询时不重复利用这些子树的计算结果，就必须分别计算每个子树实例。图4左侧展示了一个 d-DNNF，其中完全相同的子树被使用了两次。在遍历该 d-DNNF 以计算有效配置数量时，对应的子公式 $C \vee \neg C$ 需要计算两次。为避免多次计算相同子树的结果，我们像图4右侧所示那样重复利用子树。在这种情况下，对应于 $C \vee \neg C$ 的子树有两个父节点。计算出具有多个父节点的子树的满足赋值数量（此处为：$C \vee \neg C$）后，我们会缓存该值，并在其余父节点处重复利用。

部分遍历 如果我们要计算文字的部分赋值下的满足赋值数量（例如，计算某个特征的基数），只需重新计算受相应文字影响的节点的值。因此，对于每个查询，我们首先遍历 d-DNNF 中的节点，找出属于从相应文字到根节点的直接路径上的节点，并将这些节点标记出来。随后，我们再次遍历 d-DNNF，只重新计算已标记节点的计数。为了重复利用给定部分赋值不会改变的值，我们会为每个节点缓存其原始值（即未应用部分赋值时的值）。图5展示了贯穿示例中计算 B 的基数时的部分遍历（蓝色路径）。遍历只需调整$\neg B$ 与根节点 $\wedge$之间直接路径上的值。请注意，对于每个不受影响／未着色的节点， B 的基数值与其原始值相同。因此，采用部分遍历时，十二个节点中只需重新计算四个。

迭代遍历 为了简化值的重复利用，并总体减少遍历 d-DNNF 时的计算量，我们采用迭代方法，而非递归方法。节点存储在一个列表中，使得子节点总是在其父节点之前遍历。因此，处理某个节点时，其子节点的值已经可用，无需递归向下遍历。图6展示了我们所用的列表表示，对应于左侧所示的 d-DNNF。从左到右，列表表示依次展示节点的类型（即 $\wedge$）、原始计数、调整后的计数，以及指向其子节点的指针。为了在 d-DNNF 上评估查询（此处为 $\#A$），我们从列表中的第一项开始迭代遍历列表。在本例中，我们从表示文字 $\neg A$的节点开始。当到达依赖于其他节点值的节点时，所需的所有值都已经计算完毕。例如，到达第一个或节点时，我们只需查找 C 和 $\neg C$的值。

对于部分遍历，我们确保只访问已标记的节点。蓝色虚线表示在查询 $\#A$ 的第一阶段中标记的节点。随后，算法会重新计算这三个已标记节点各自的值。查询 $\#A = 2$ 的结果随后会保存到根节点的临时值中。

部分计算 在部分遍历期间计算节点的值时，可能只有少数子节点发生变化。在这种情况下，与其重新计算整个节点，不如调整缓存值。例如，对于一个有十个子节点的$\wedge$节点，如果只有一个子节点 c' 发生变化，我们就不计算这十个子节点的乘积，而是根据发生变化的值调整原始值，如公式1所示。如果超过半数的子节点发生变化，我们就按照算法1重新计算该值。否则，对于每个发生变化的子节点，我们先将缓存值除以 c_{old} 的旧值，再乘以其新值。类似地，我们将缓存值减去的旧值，然后

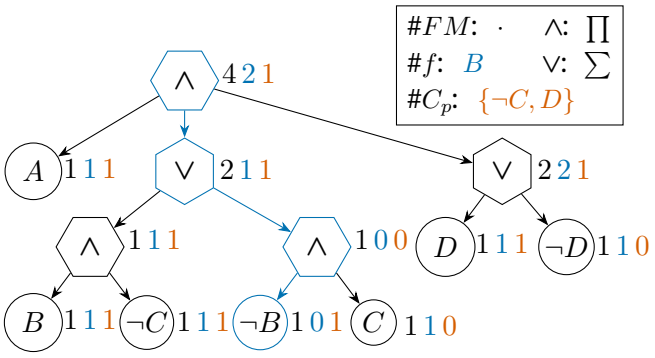

图 5：针对特征 B 基数的 d-DNNF 部分遍历

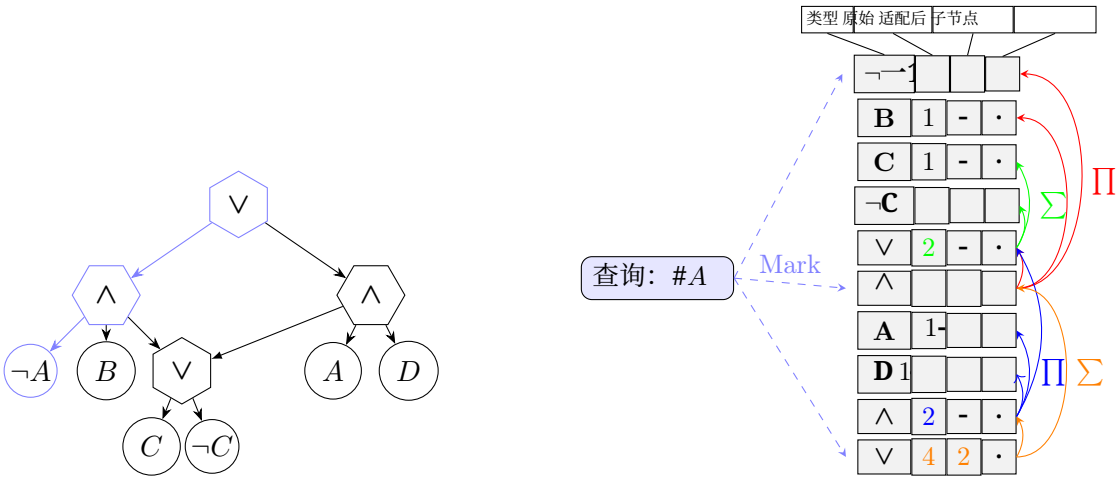

图6：表示图4的迭代遍历列表格式

将 ∨ 节点的新值加上去，如公式2所示。每个发生变化的节点需要进行两次算术运算。例如，重新计算一个∧节点需要进行一次乘法和一次除法。因此，只有当发生变化的子节点值少于一半时，我们才进行部分计算。

$$\#node(\wedge) = \begin{cases} \prod_{c \in children} \#c_{new} & \geq \frac{n}{2} \text{ children changed} \\ \frac{\#\wedge_{old} \cdot \prod_{c \in adapted} \#c_{new}}{\prod_{c \in adapted} \#c_{old}} & \text{else} \end{cases} \tag{1}$$

$$\#node(\vee) = \begin{cases} \sum_{c \in children} \#c_{new} & \geq \frac{n}{2} \text{ children changed} \\ \#\vee_{old} + \sum_{c \in adapted} \#c_{new} - \#c_{old} & \text{else} \end{cases} \tag{2}$$

核心特征与无效特征 核心特征和无效特征分别出现在所有有效配置和所有有效配置中都不出现 [1]。因此，如公式3所示，无需再次遍历 d-DNNF，即可直接得出核心特征和无效特征的基数。对于核心特征，其满足赋值数量始终等于总体满足赋值数量。无效特征的基数始终为零。作为对 ddnnife 的优化，我们提出利用 d-DNNF 的平滑性预先计算核心特征和无效特征，从而有望减少所需的遍历次数。通过这一新方案，我们只需遍历一次文字节点，就能根据以下性质识别所有核心特征和无效特征。特征 f 为核心特征，当且仅当 d-DNNF 中只出现正文字。类似地，特征 f 为无效特征，当且仅当只出现负文字。在贯穿示例图 2 中，特征 A 为核心特征，因为平滑 d-DNNF 中只出现正文字。

$$\#f = \begin{cases} \#FM & \text{if } f \text{ is core} \\ 0 & \text{if } f \text{ is dead} \\ \#f & else \end{cases} \tag{3}$$

下面简要说明我们检测核心特征与死特征方法的正确性。假设我们用于计算部分配置基数的算法是正确的，那么如以下说明所示，我们采用核心/死特征的方法的正确性便可直接推出。我们先说明核心特征的情况。如果变量 v 只出现正文字，那么使用算法 2 计算 $\#v$ （即 v 的基数）所得结果始终与无任何假设时的 d-DNNF 相同。唯一可能导致结果变化的情况是出现与我们的假设相冲突的 v 的负文字。因此，满足赋值的数量

在其中出现的 v，等于使 v 成为核心特征的所有满足赋值的总数。对于死特征，情况也类似。如果变量w 只出现负文字，那么使用算法 2 计算 #($\neg w$) 始终会得到与无假设时的 d-DNNF 相同的结果。因此，在这种情况下，不存在包含 w 的满足赋值，否则结果就会不同。

关于核心特征与无效特征的信息也可用于减少部分配置所需的查询次数。每当部分配置中包含无效特征或排除核心特征时，该配置便不可满足，因此无需通过 d-DNNF 遍历进行评估。此外，在部分遍历的标记阶段，可以忽略已包含的核心特征和已排除的无效特征。

总结 所介绍的六种优化分别减少了所需的查询次数，或降低了单次遍历的工作量。这两方面的影响都有可能提升重复利用 d-DNNF 来计算特征或部分配置的基数时的效率。在第 6 节中，我们将考察该方法的整体性能，并进一步分析特定优化的效果。

5 工具支持：ddnnife

我们介绍的算法和优化已在开源 d-DNNF 推理器 ddnnife 中实现；它是重复利用 d-DNNF 的瑞士军刀。ddnnife 使用 Rust[3] 实现，因为 Rust 通常能够编译出高效的目标代码，并且在提升内存安全性方面提供了出色的支持。源代码和示例输入数据可在 GitHub[4] 获取，采用 LPGLv3[5] 许可证，潜在用户即使在商业场景中也只受少量限制即可使用 ddnnife。ddnnife 接受符合现有标准格式（参见第 5.1 节）的 d-DNNF 作为输入。解析 d-DNNF 时，可使用 ddnnife 对其进行平滑（如有必要），并计算特征模型、特征或部分配置的基数。在第 6 节中，我们使用 ddnnife 等工具评估重复利用 d-DNNF 的优势。

5.1 解析 d-DNNF

输入格式ddnnife 支持 c2d[6] [37]设定的标准格式，以及 d4 [39]引入的格式变体[7] 。由于支持这两种格式，ddnnife 能够解析各种流行 d-DNNF 编译器生成的 d-DNNF，包括 c2d [37], 、dSharp [38],和 d4 [39]。

列表 1 展示了我们贯穿示例图 2 的 c2d 格式。[8] 第一行 nnf v e n 表示节点数 (v)、边数 (e) 和变量数 (n)。其余每一行都描述 d-DNNF 中的一个节点。L 1 表示引入变量 1 的一个正文字。与节点由行 A n i j k 描述，其中 n 是子节点数，i、j、k 是包含子节点的行的索引。请注意，包含节点的第一行索引为零。或节点还带有一个附加参数，用于指示决策变量（即在左侧子节点中始终为真、在右侧子节点中始终为假的变量）。O d n i j k 描述一个决策变量为 d、包含 n 个子节点且索引为 i、j、k 的或节点。决策变量 d 为 0 表示不存在决策变量。

列表 2 展示了贯穿示例图 2 的 d4 格式。[9] 在此，每一行要么描述一个节点，要么描述节点之间的一条边，并以 0 结尾。描述节点的行

[3]https://www.rust-lang.org/[4]
https://github.com/SoftVarE-Group/d-dnnf-reasoner/releases/tag/March2023[5]
https://www.gnu.org/licenses/lgpl-3.0.de.html[6]http://reasoning.cs.ucla.edu/c2d/[7]
https://github.com/crillab/d4[8]http://reasoning.cs.ucla.edu/c2d/[9]
https://github.com/crillab/d4

总是先指定节点的类型，即或（o）、与（a）、真（t）、假（f），然后指定该节点的索引。与 c2d 格式不同，这些索引并非指行号，而是指为每个节点指定的索引。边的格式始终为 p c x y z 0，其中 p 是父节点，c 是子节点，x y z 是任意数量的文字，沿此路径时这些文字均设为真。例如，第 4 行指定索引为 1 的节点有一个索引为 3 的子节点（Φ_3）。此外，文字 2 和 -3 合取后得到子公式（$\Phi_3 \wedge (2 \wedge -3)$）$\vee \Phi_{rest}$，其中 Φ_{rest} 是从节点 1 出发的其他边所产生的公式。

列表 1：c2d 格式

```
nnf11 11 4
0: L 1 # A
1: L 2 # B
2: L -3 # !C
3: L -2 # !B
4: L 3 # C
5: L 4 #D
6: L -4 # !D
7: A2 1 2
8: A2 3 4
9: O0 2 7 8
10: O4 2 5 6
11: A3 0 9 10
```

列表 2：d4 格式

```
o1 0
o2 0
t 30
a4 0
1 3 2 -3 0
1 3 -2 3 0
2 3 4 0
2 3 -4 0
4 3 1 0
4 0 2 0
4 0 1 0
```

数据结构 图 7 展示了 ddnnife 用于存储 d-DNNF 的数据结构。每个 d-DNNF 主要由一组节点组成，并附带一些用于简化算法的额外信息。这些额外信息包括：(1) 指向文字节点索引的指针，(2) 核心文字集合，以及 (3) 一个列表，用于标记部分遍历期间需要重新计算的节点。节点具有 NodeType 枚举，用于指示 d-DNNF 节点的类型，即与、或、文字、真和假。每个节点都包含指向所有父节点的指针，以便在标记部分遍历时从底部开始遍历。为了复用值，我们为每个节点附加了原始计数（即不带假设时的计数）以及当前查询下的临时值。与节点和或节点还提供了一组指向其子节点的索引。

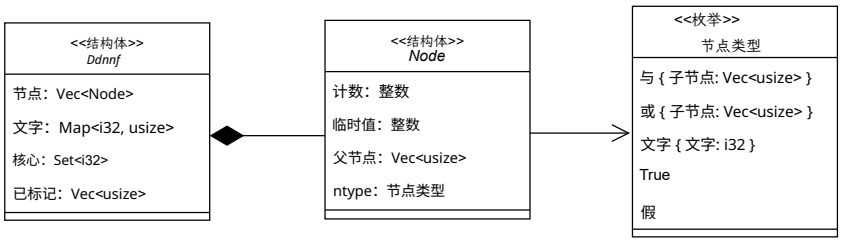

图 7：ddnnife 数据结构

预处理 在对特征或部分配置进行查询之前，ddnnife 会执行一些预处理步骤，以简化这些查询的计算。首先，必要时通过添加平滑节点使给定的 d-DNNF 平滑（参见第 4.2 节）。其次，添加从子节点指向父节点的连接，以简化部分遍历的标记算法。第三，保存指向文字节点索引的指针，以供标记时作为起点。第四，通过检查某个变量是否

仅以正文字（核心特征）或负文字（死特征）的形式出现，分别识别核心特征与无效特征。第五，在不作任何假设的情况下，计算每个 d-DNNF 节点的模型计数。之后，部分遍历会重复使用这些值。完成各项预处理步骤后，存储得到的 d-DNNF 表示，并将其用于查询。

5.2 算法的实现

ddnnife 支持第 4.1 节中介绍的用于计算特征模型、特征和部分配置基数的算法。由于解析步骤确保每个 d-DNNF 都是平滑的，因此这些算法在计算过程中无需跟踪变量。算法1的实现经过了轻微调整，以纳入 $\top$ 和 $\bot$ 节点，因为 dSharp [39] 和 d4 [38] 在某些情况下会使用它们。 $\top$ 和 $\bot$ 的处理方式类似：文字 $\top$ 的值始终为 1，而 $\bot$ 的值始终为 0。

优化ddnnife 包含第 4.2 节中介绍的所有优化。不过，根据将 ddnnife 应用于表示特征模型的 d-DNNF 时的观察结果，部分优化做了轻微调整。

折叠仅应用于 $\wedge$ 节点，原因有二。首先，乘法的计算开销高于加法，因此减少加法次数所带来的运行时间收益较小。其次，在实践中，由 c2d、dSharp 和 d4 构造的 d-DNNF 中，$\vee$ 节点通常恰好有两个子节点。对于恰好有两个子节点的情况，折叠优化没有任何收益，因为即使只有一个子节点发生变化，仍然需要进行两次算术运算。

处理大型部分配置（即包含或排除许多特征）时，d-DNNF 中很大一部分会发生变化，因此需要遍历。如果部分配置的大小超过某个阈值，标记节点以便重新计算所带来的开销就不再值得。此时，我们不会使用部分遍历，而是重新评估整个 d-DNNF。

5.3 用法

在进行不同的分析之前，ddnnife 始终要求输入一个 d-DNNF，格式可以是 c2d 或 d4。在本节中，我们简要介绍 ddnnife 的不同使用模式，即针对给定的一组查询进行一次性计算，以及用于交互式使用的流式模式。有关语法和技术特性的更多详情，请参阅代码仓库。[10]

一次性计算 将 d-DNNF 用于一次性计算时，用户有三种方式来指定查询：(1) 计算某个特征的基数；(2) 计算某个部分配置的基数；(3) 计算所有特征的基数；或者 (3) 查询文本文件中给出的一组不同查询。调用后，该工具会以 .csv 或命令行输出的形式提供结果，然后终止。

流式模式 如果无法预先获知所有查询，ddnnife 的流式模式可用于交互式查询。启动时，ddnnife 会一次性加载并预处理 d-DNNF，然后等待有关特征或部分配置的查询。流式模式遵循严格的协议，用户可通过基于协议的交换，将 ddnnife 集成到非 Rust 工具中。不过，ddnnife 也可通过人工输入进行交互式使用。加载完成后，用户可以查询 d-DNNF（例如，输入 count v 1 来计算变量 1 的基数）。有关协议的详情，请参阅代码仓库。[11]

[10]https://github.com/SoftVarE-Group/d-dnnf-reasoner/releases/tag/March2023[11]
https://github.com/SoftVarE-Group/d-dnnf-reasoner/releases/tag/March2023

选项 除了使用模式和用于指定查询的参数外，ddnnife 还提供了一些额外参数来控制其行为。预处理，尤其是对 d-DNNF 进行平滑化，通常需要耗费相当多的初始工作。因此，我们提供了一个选项，可保存平滑后的 d-DNNF 以供日后使用。用户还可以启用多线程，以并行处理多个查询。在某些情况下，d4 会在 d-DNNF 中省略完全可选特征。忽略这些变量会引发两个问题。首先，由于这些变量可以任意赋值，满足赋值的数量会相差 2 的省略变量数次方。其次，ddnnife 无法计算这些特征的基数，也无法计算包含或排除其中任一特征的部分配置的基数，因为它们并未包含在 d-DNNF 中。因此，我们提供了一个参数来声明被省略的变量，以避免产生错误结果。

摘要 对于给定的 d-DNNF，ddnnife 可供交互式使用，也可通过一次性计算来执行重复计数查询。借助 d4 [39], c2d [37],和 dSharp [38], 等常用编译器，我们可以将特征模型转换为 d-DNNF，再用 ddnnife 执行特征模型计数。在第 6 节中，我们考察了 ddnnife 及其优化在特征模型上的性能。为此，我们重点关注一次性计算模式。

6 评估

在实证评估中，我们分析了与当前最佳水平相比，复用 d-DNNF 来计算特征基数和部分配置基数所带来的优势。将 d-DNNF 用于特征模型计数是否可行，取决于编译（离线阶段）的可扩展性，以及使用推理器复用已编译 d-DNNF（在线阶段）的可扩展性。因此，我们在实证评估中同时考察这两个阶段。我们提供了一个公开可用的复现包[12] ，其中包含所有评估工具、研究对象系统（不包括行业合作伙伴的九个保密系统）以及测量所需的工具。

6.1 研究问题

通过提出这些研究问题，我们旨在考察与重复进行 #SAT 调用相比，重复利用 d-DNNF 的优势。此外，我们还比较了公开可用工具的性能。

研究问题1 将 d-DNNF 编译器应用于工业特征模型时，效率如何?

要将我们的算法应用于工业特征模型，首先必须能够将相应的公式编译为 d-DNNF。通过研究问题1，我们考察并比较了 d-DNNF 编译器在运行时间和生成的 d-DNNF 大小方面的可扩展性。我们也考虑了大小，因为 ddnnife 的运行时间取决于 d-DNNF 的大小。

研究问题2 哪些工具及工具组合能够计算出正确的基数?

为检验所考虑工具的正确性，我们比较了计算所得的基数。具体而言，我们通过实证验证了所提出的 ddnnife 的正确性。

研究问题3 与反复进行 #SAT 调用相比，复用 d-DNNF 来计算特征和部分配置的基数有多高效?

在通过研究问题1和研究问题2考察将 d-DNNF 用于特征模型计数的一般适用性之后，我们通过研究问题3考察本文的主要关注点（即重复利用 d-DNNF 进行特征模型计数的优势）。我们将基于 d-DNNF 的方法的运行时间

[12]https://github.com/SoftVarE-Group/exploiting-ddnnfs-eval/releases/tag/2023年3月

与通过重复调用 #SAT 计算特征基数和部分配置的运行时间进行比较。在此，我们分别考虑计入和不计入编译时间的 d-DNNF 重用情况。此外，我们还考察 d-DNNF 利用需要进行多少次计算才能达到相较于重复调用 #SAT 的盈亏平衡点。

研究问题 4 ddnnife 中的优化在多大程度上提升了运行时间效率?

尽管所提出的基础算法针对每个查询所需的时间仅与 d-DNNF 节点数呈线性关系，但对于大型 d-DNNF，基础算法仍可能需要对数百万个节点执行算术运算。我们的优化（参见第 4.2 节）旨在减少每次查询所需的工作量以及查询次数。针对研究问题 4，我们考察所提出的优化对 ddnnife 运行时间的影响。

6.2 实验设计

研究对象系统 在实证评估中，我们只考虑工业特征模型，不考虑人工生成的特征模型。正如先前研究所观察到的，人工生成的特征模型可能导致运行时间无法代表现实情况 [44, 62]。通过选择这些研究对象系统，我们旨在广泛覆盖来自不同领域的工业特征模型。为此，我们主要采用相关工作基准测试中使用的公开特征模型。表 1 概述了各研究对象系统的领域、特征数、子句数和基数。

首先，我们纳入了 Knüppel 等人提供的基准测试[13] 中的特征模型 [10] ，其中包含各种 CDL[14] 模型、KConfig[15] 模型以及一个汽车产品线。对于这 117 个 CDL 特征模型，我们在先前的工作 [43] 中发现，考虑到特征数量、约束数量以及 #SAT 求解器的性能等多项指标，它们彼此高度相似。为避免产生偏差，我们决定在基准测试中只纳入三个 CDL 特征模型，而不是全部 117 个。具体而言，我们选取了特征数量最少、居中和最多的特征模型。其次，我们的基准测试纳入了 Oh 等人提供的系统软件和应用领域[16] 中的特征模型 [63]。第三，我们纳入了 Pett 等人提供的 BusyBox 特征模型[17] 的一个版本 [64]。第四，我们考察了从金融服务领域[18] 提取的一个特征模型 [65, 66]。第五，我们考察了 Kowal 等人发表的另一个来自汽车领域的特征模型 [67]。第六，我们纳入了一个可在 FeatureIDE 示例中公开获取的数据库特征模型[19]。第六，我们纳入了来自一次产业合作、代表汽车产品线的九个特征模型，但遗憾的是，这些模型无法公开。总体而言，我们的基准测试包含 62 个工业特征模型。

评估工具 在选择工具时，我们的目标是将 d-DNNF 编译与复用同特征模型计数的最先进水平进行比较。目前，最具可扩展性且普遍适用的解决方案似乎是将特征模型转换为合取范式，并将 #SAT 求解器作为黑盒调用 [25, 68, 69]。为获得有代表性的结果，我们力求纳入所有公开可用的 d-DNNF 编译器 [37–39] （即将合取范式编译为 d-DNNF 的工具）和推理器（即使用 d-DNNF 计算满足赋值数量的工具），以及速度最快的公开可用现成的 #SAT 求解器 [25, 28, 53, 54]。

[13]https://github.com/AlexanderKnueppel/is-there-a-mismatch[14]https://ecos.sourceware.org/[15] https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html[16]https://github.com/jeho-oh/Smarch[17] https://github.com/PettTo/Measuring-Stability-of-Configuration-Sampling[18]https://github.com/PettTo/SPLC 2019_The-Scalability-Challenge_Product-Lines[19] https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/ featureide_示例

表 1：研究对象系统概览

领域	模型	特征	子句	基数
操作系统	37	94–62,482	190–343,944	10^{10}–10^{417}†
其他软件*	13	11–31 ,012	1–102 ,705	10^{3}–10^{5}†
汽车	11	384–18 ,616	1,020–350 ,221	10^{4}–10^{1534}†
金融服务	1	771	7,241	10^{13}

* 包括归档器（2）、数据库（2）、视频（2）、网络（1）、垃圾回收（1）、图像（1）、编译器（1）、求解（1）、网页开发（1）和团队管理（1）

† 由于部分模型的求解器超时，基数数据不完整

表 2 概述了我们找到的工具。在一份关于 2020 年模型计数竞赛（MC2020）的报告中，Fichte 等人 [29] 列出了能够计算满足赋值数量的现有工具，其中包括 d-DNNF 编译器。在我们的实证评估中，我们纳入了他们列表中的每个 d-DNNF 编译器，即 c2d [37], 、dSharp [38], 和 d4 [39]。据我们所知，还没有 d-DNNF 推理器能够复用 d-DNNF 处理重复查询，并且经过实证评估或在其他出版物中被考察。不过，我们将搜索范围扩展到未发表科学出版物的工具，找到两个提供此类功能的工具：Caridriot 等人开发的推理器 query-ddnnf[20] ，以及 GitHub 上 d4 的一个新版本。与原始工作 [39]相比，扩展后的 d4 版本不支持基于已有 d-DNNF 进行推理，但能够针对给定的合取范式计算多个查询的满足赋值数量；我们也将其纳入评估（称为 d4-query）。我们将方案 ddnnife、query-ddnnf 和 d4-query 作为基于 d-DNNF 的推理器进行评估。在初步实验中，我们发现将 query-ddnnf 与 dSharp 结合使用时，得到的基数总是为零，这可能是因为 dSharp 在某些情况下违反了 c2d 规定的标准。因此，我们排除了 query-ddnnf 与 dSharp 的组合。query-ddnnf 不支持最新 d4 版本的输出格式。因此，我们使用较早的版本评估 query-ddnnf。[21]

为评估我们采用 d-DNNF 的方法，我们将 #SAT 求解器 sharpSAT、countAntom 和 Ganak 纳入基线。在近期的工作中 [43], ，我们将这三个求解器确定为分析特征模型时速度最快的 #SAT 求解器。此外，Ganak 赢得了模型计数竞赛 2020 [29],，而 sharpSAT 则在 2021年 和 2022 中获胜。[22]

只有 countAntom 和 ddnnife 支持多线程。为便于与其他工具进行比较，我们决定使用单线程评估每种工具。由于所考察的每个 d-DNNF 编译器和 #SAT 求解器都要求以合取范式作为输入格式，因此我们在实验前使用 FeatureIDE 将每个特征模型转换为合取范式 [12] 。

实验 1：编译为 d-DNNF 在第一个实验中，我们测量将表示研究对象系统的合取范式转换为 d-DNNF 所需的运行时间。为此，我们在全部 62 个研究对象系统上评估每个 d-DNNF 编译器，即 c2d、dSharp 和 d4，每个特征模型的超时时间为 30 分钟。为了对编译所需的运行时间有更直观的认识，我们还评估了使用三个 #SAT 求解器（即 sharpSAT、countAntom 和 Ganak）计算有效配置数量所需的运行时间。根据初步实验的结果，我们将超时时间设为五分钟。此外，#SAT 求解器若超过五分钟的超时时间，就极不可能适用于实验 2 和实验 3 中的重复查询。对于编译器与研究对象系统的每种组合，我们重复实验 50 次。编译得到的 d-DNNF 用于实验 2 和实验 3。实验 1 得出的见解用于

[20]https://www.cril.univ-artois.fr/KC/d-DNNF-reasoner.html[21] https://www.cril.univ-artois.fr/KC/d4.html[22] https://mccompetition.org/index.html

表 2：评估工具概览

求解器	Type	参考文献
countAntom	#SAT 求解器	[53]
Ganak	#SAT 求解器	[54]
sharpSAT	#SAT 求解器	[28]
c2d	d-DNNF 编译器	[37, 55]
d4	d-DNNF 编译器 [39]	
dSharp	d-DNNF 编译器 [38]	
ddnnife	d-DNNF 推理器	本次提交
query-ddnnf	d-DNNF 推理器	无科学出版物
d4-query	d-DNNF 推理器	无科学出版物

回答研究问题1。

实验 2：特征基数 在第二个实验中，我们测量计算给定特征模型中所有特征基数所需的运行时间。为此，我们在每个研究对象系统上评估每种 d-DNNF 工具组合（由编译器和推理器组成）以及 #SAT 求解器。我们将计算单个特征模型中所有特征基数的超时设为 30 分钟。每项测量重复 50 次。实验 2 的结果用于回答研究问题2–研究问题4。

实验 3：部分配置的基数 在第三个实验中，我们测量计算给定特征模型的部分配置基数所需的运行时间。我们为每个研究对象系统随机生成 250 个部分配置。我们确保每个部分配置都可满足，因为我们假设，可满足的配置更能反映文献中相关应用所体现的特征模型计数的实际用法 [17]。为生成部分配置，我们迭代地随机选择一个特征，并将其选中或取消选中。如果由此产生无效配置，我们就舍弃当前选择并继续操作。将这 250 个配置分成 50 组，每组的特征数分别为 2、5、10、20 和 50（随机纳入或排除）。有 11 个特征模型的特征数少于 50；对于这些模型，我们仅生成特征数少于相应特征模型的配置集。通过采用不同的配置规模，我们旨在涵盖广泛的、取决于部分配置基数的分析 [17]。与实验 2 一样，我们在每个研究对象系统上评估每种 d-DNNF 工具组合以及 #SAT 求解器。我们将计算单个特征模型的全部 250 个部分配置基数的超时设为 30 分钟。每项测量重复 50 次。实验 3 的结果用于回答研究问题2–研究问题4。

实验 4：ddnnife 优化在第四个实验中，我们考察了第 4.2 节中介绍的优化所产生的影响。具体而言，我们比较了启用不同优化的 ddnnife 变体。表3列出了我们用于评估的 ddnnife 变体。首先，我们考虑一种不采用任何优化的朴素方法：针对每个查询，递归遍历整个 d-DNNF，且不复用任何子树。第二种变体同样针对每个查询递归遍历 d-DNNF，但会复用子树。对于第三、第四和第五种变体，我们分别禁用一项优化：部分遍历、部分计算以及核心/死特征。作为基线，我们采用先前实验中使用的完全优化版 ddnnife，作为第六种变体。对于每种 ddnnife 变体，我们重复实验2和实验 3 的实验设计，以衡量这些优化对计算特征基数和部分配置的影响。

表3：ddnnife 优化变体

Name	启用的优化
朴素	None
重复利用子树	子树的重用
无部分遍历	除部分遍历外的所有内容
无部分计算	除部分计算外的所有内容
无核心/非核心	除核心/死特征的使用外
ddnnife	All

在此过程中，我们使用相同的输入 d-DNNF、输入部分配置和超时时间。除了重复这些实验外，我们还评估了这些优化在不可满足的部分配置上的效果，因为有些优化实际上无法应用于不可满足的部分配置。实验 4 得出的见解用于回答研究问题 4。

统计检验 我们使用曼-惠特尼显著性检验 [70] ，基于 50 次重复实验比较两种工具在特征模型上的性能，因为样本彼此独立，且我们不假设数据服从正态分布。在此，我们始终假设显著性水平为 $\alpha = 5\%$。为考察显著结果的效应量，我们使用科恩 d [71]。效应量的评级采用表 4 所示的分类，该分类由 Sullivan 和 Feinn [71]提出。例如，值为 $0.5 \leq d < 0.8$ 时，效应量为中等。

科恩 d	效应量分类
$0.0 \leq d < 0.2$	极小
$0.2 \leq d < 0.5$	小
$0.5 \leq d < 0.8$	中等
$0.8 \leq d < 1.3$	大
$1.3 \leq d$	极大

表 4：科恩 d 分类效应量，改编自 Sullivan 和 Feinn [71]

技术配置整个实证评估均在一台运行 Ubuntu 20.04.3 LTS、采用 64 位架构的服务器上进行。处理器为 Intel(R) Xeon(R) CPU E5-260v3，时钟频率为 2.40 GHz。总计可用内存为 256 GB。测量期间，该机器仅用于我们的评估，以减少其他进程的影响。对于每种工具，内存上限均设为 8 GB，因为我们认为在实际使用中，8 GB 是硬件（例如笔记本电脑或个人电脑）的合理上限。此外，在初步实验中，我们发现进一步提高内存上限并不会使测得的运行时间出现明显差异。为测量运行时间，我们使用了一个基于 Python 的框架，该框架在内部使用 timeit 模块。[23]

6.3 结果

实验 1 图8展示了各 d-DNNF 编译器转换每个研究对象系统所需的运行时间。x 轴上的每个点对应一个研究对象系统。y 轴以对数刻度显示编译器进行 50 次重复运行的平均运行时间和标准差。在 62 个研究对象系统中，有 55 个至少能由一个 d-DNNF 编译器成功编译

[23]https://docs.python.org/3/library/timeit.html

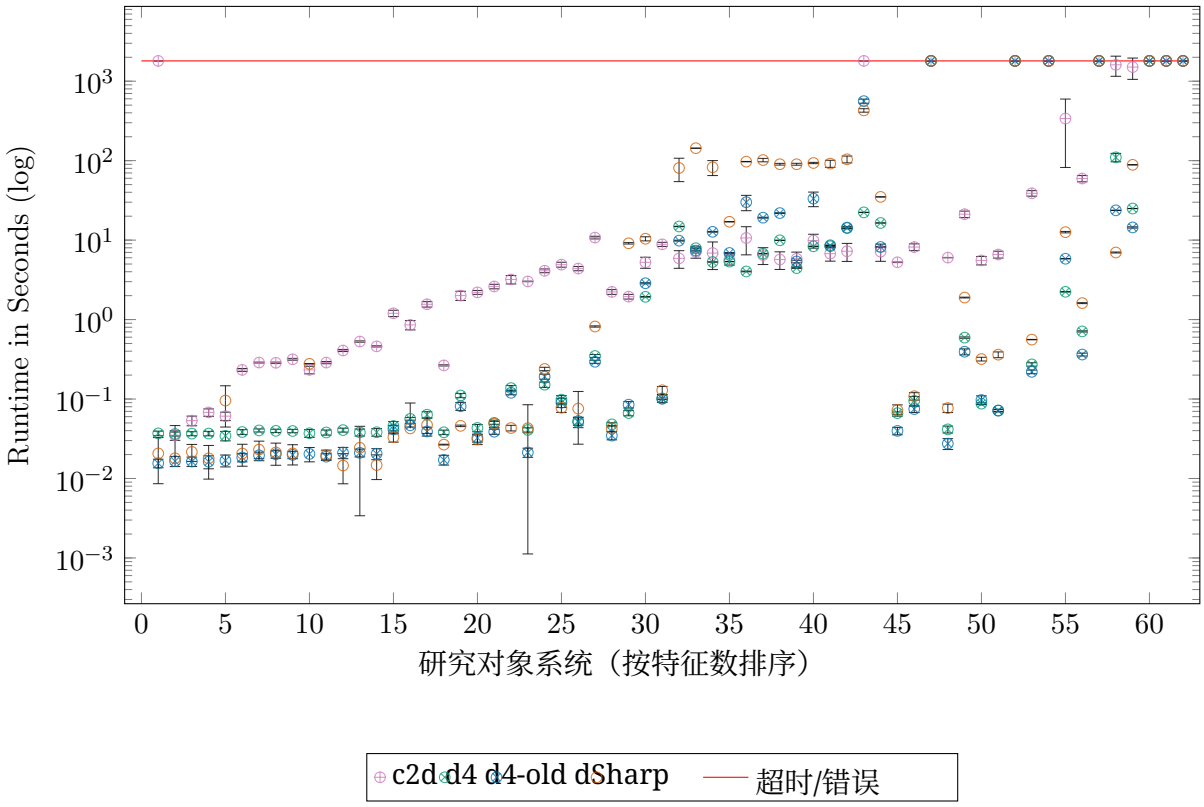

图8 ：d-DNNF 编译器离线阶段的运行时间（平均值与标准差）

合取范式为 d-DNNF，且用时不超过 30 分钟。对于相同的七个特征模型，三个 #SAT 求解器均达到五分钟的超时限制（参见第6.2节）。[24] 52 个研究对象系统至少能由一个编译器在 10 秒内完成编译。d4（两个版本）和 dSharp 均成功编译了 55 个特征模型，而 c2d 成功编译了 53 个。对于 LLVM，一个仅包含单个单文字子句的研究对象系统会导致 c2d 抛出错误，这可能是由对单文字子句进行预处理后留下非法（空）公式所致。对于每个编译器都成功处理的 53 个模型，d4、dSharp 和 c2d 的平均耗时分别为 4.62 秒、21.5 秒和 69.5 秒。大多数（83%）d-DNNF 所需空间不足 10 MB。最快的编译器 d4 生成的 d-DNNF 至多占用 20.1 MB（中位数为 0.04 MB）。

实验2 除 dSharp 和 sharpSAT 外，所有工具都为所有纳入考察的特征模型中的每个特征计算出了相同的结果（即基数）。dSharp（作为黑盒）和 sharpSAT 的结果分别有 55.28%（涉及 31 个模型）和 15.32%（31 个）与计算出的总基数不一致。对于由 dSharp 编译的 d-DNNF，ddnnife 的结果有 0.54% 不一致。

图9展示了使用 d-DNNF（计入离线阶段和在线阶段）以及重复调用 #SAT 计算所有特征的基数所需的运行时间。对于每个成功编译为 d-DNNF 的特征模型（62 个中的 55 个），至少有一种 d-DNNF 工具组合能够在 30 分钟内成功计算出所有特征的基数。#SAT 求解器仅对 49 个特征模型计算出了所有特征的基数。对于每个特征模型，ddnnife 与 d4 结合的速度都显著快于每个 #SAT 求解器（$p < 7.1 \cdot 10^{-18}$），且效应量极大（$d \geq 8.5$）。

表5展示了三种 #SAT 求解器在 30 分钟超时限制内能够评估的特征相对比例。虽然最快的 d-DNNF 工具组合（ddnnife

[24]四种 Linux 变体、Embtoolkit、Freetz 和 Automotive05。

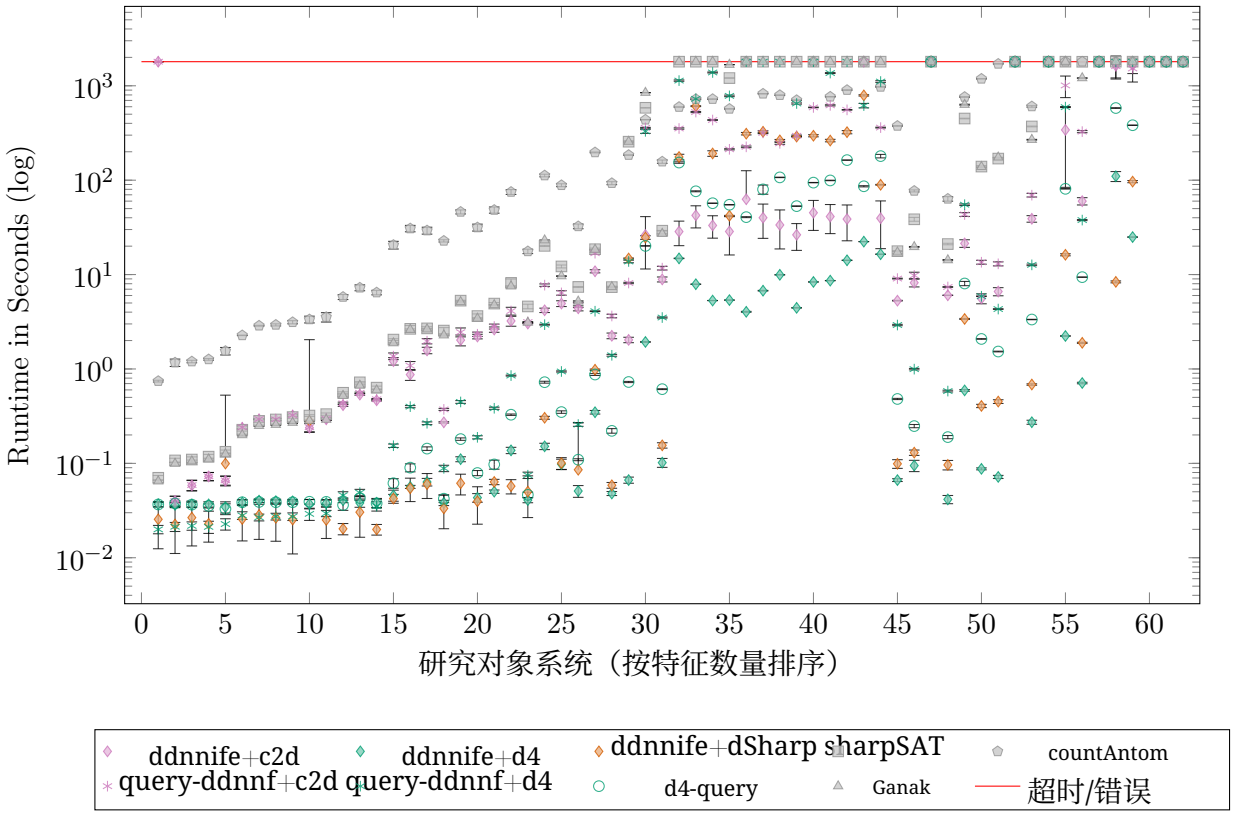

图9：特征基数的离线 + 在线阶段运行时间（平均值与标准差）

和 d4）评估*Embtoolkit* 的特征基数仅需 48.3 秒，但最快的 #SAT 求解器（sharpSAT）用 30 分钟也只评估了 0.86% 的特征。根据运行时间进行外推，sharpSAT 需要 58 小时才能评估*Embtoolkit* 的所有特征，比 ddnnife 与 d4 的组合慢 8,316 倍（不计编译时间则慢 39,253 倍）。同样根据此类外推，最快的基于 d-DNNF 的工具组合平均比最快的 #SAT 求解器快 480 倍。对于特征数超过 2,000 的特征模型，最快的基于 d-DNNF 的工具组合平均甚至快 2,307 倍。若按每个实例选取最快的工具，使用 d-DNNF 时，全部 55 个特征模型均可在 2.8 分钟内完成评估；而使用 #SAT 则需要 67小时 （慢 1,421 倍）。考虑编译时间以及评估单个特征的平均计算时间后，最快的 d-DNNF 编译器与推理器组合，在查询数达到 3.9 次（中位数）后，便可与最快的 #SAT 求解器持平。对于所有评估时间超过 0.1 秒的特征模型，ddnnife 与 d4 的组合均显著快于任何不包含 ddnnife 的工具或工具组合（$p < 7.1 \cdot 10^{-18}$），且效应量极大（$d \geq 4.86$）。在在线阶段，若分别为两种推理器选取配有编译器的最快组合，则 ddnnife 评估所有特征模型需要 4.3 分钟，而 query-ddnnf 需要 72 分钟。采用部分遍历（参见第 4.2 节）后，平均而言，ddnnife 每个特征只需重新计算 d-DNNF 节点的 6.5%（由 d4 编译）、4.4%（由 dSharp 编译）和 10.1%（由 c2d 编译）。

实验 3 除 dSharp 外，所有工具针对每个部分配置和特征模型计算出的基数都相同。dSharp（黑盒）和使用 dSharp 的 ddnnife 的结果分别在 41.83%（涉及 31 个模型）和 5.83%（涉及 31 个模型）的情况下与这些结果不同。

图 10 展示了计算所生成的 250 个

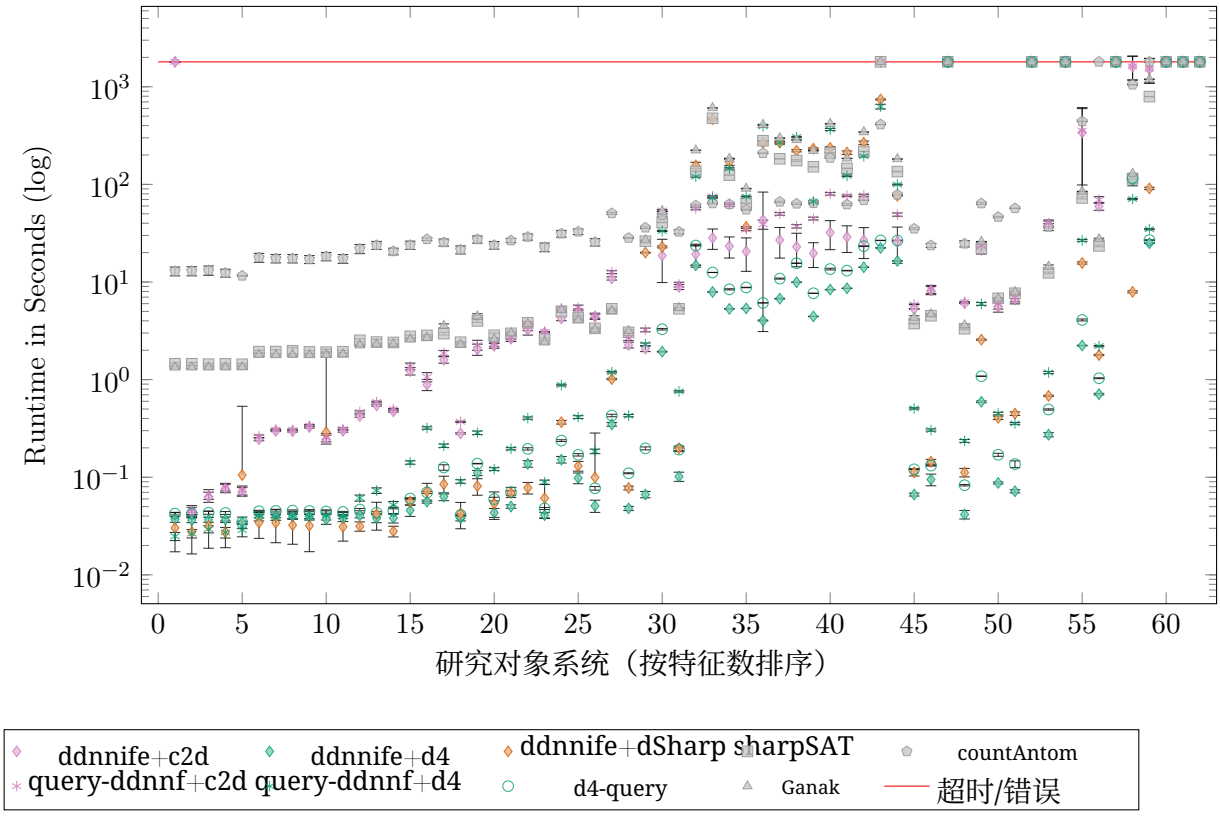

图 10：部分配置基数的离线 + 在线阶段运行时间（平均值和标准差）

部分配置的基数所需的运行时间，采用的方法是 d-DNNF 和重复的 #SAT 调用。对于每个成功编译的 d-DNNF（共 55 个），ddnnife 和 query-ddnnf 与任意编译器组合，以及 d4-query，均能在 30 分钟内计算出这 250 个部分配置的基数。至少有一个 #SAT 求解器成功评估了 53 个特征模型的部分配置。对于所有成功评估的特征模型，ddnnife 与 d4 的组合显著快于（$p < 7.1 \cdot 10^{-18}$）最快的 #SAT 求解器，且效应量极大（$d \geq 8.6$）。对于每个特征模型，最快的基于 d-DNNF 的工具组合比最快的 #SAT 求解器快 4.1 至 394 倍。总体而言，最快运行时间分别为 d-DNNF 的 3.7 分钟和 #SAT 的 44 分钟（慢 11.9 倍）。采用 ddnnife 的最快工具组合评估所有实例需要 5.23 分钟，而 d4-query 和与 query-ddnnf 的组合分别需要 5.88 分钟和 24 分钟。

表5：#SAT 求解器已评估特征的占比

系统	countAntom	Ganak	sharpSAT
se77x	98.1%	21.3%	47.23%
integrator_arm9	91.7%	23.4%	31.3%
FreeBSD	46.1%	4.87%	5.73%
Automotive07a	5.47%	30.2%	36.1%
Automotive02	2.28%	17.5%	19.6%
Embtoolkit	0.09%	0.65%	0.86%

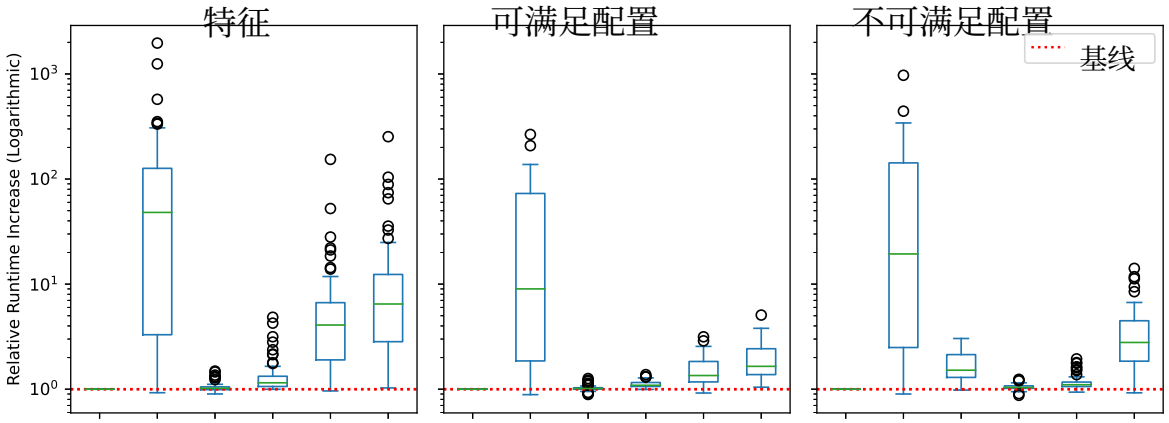

图 11：d4 d-DNNF 上 ddnnife 变体的运行时间 NNFs

实验 4 对于实验 4，我们仅展示 ddnnife 在由 d4 编译的 d-DNNF 上的运行时间图，因为 dSharp 和 c2d 生成的 d-DNNF 结果非常相似。不过，c2d 和 dSharp 的结果也包含在我们的复现包中。图 11 展示了不同 ddnnife 变体相较于完全优化变体的运行时间相对增幅。例如，数值为 1,000 表示对于给定的特征模型，某个 ddnnife 变体的运行速度比完全优化变体慢 1,000 倍。图中分别展示了计算特征基数（左）、可满足配置基数（中）和不可满足配置基数（右）所需的运行时间。

对于特征计数，启用所有优化的 ddnnife 运行时间最短，在全部 55 个特征模型上的总耗时为 5.29 分钟。禁用核心/死特征优化和部分计算优化后，运行时间相近，分别为 5.31 分钟和 6.40 分钟。不使用部分遍历时，ddnnife 评估这 55 个特征模型需要 30.0 分钟，比完全优化版本慢 5.67 倍。完全朴素的变体未使用第 4.2 节介绍的任何优化，在 55 个特征模型中有 13 个触发了 30 分钟超时。其他变体均未触发超时。对于评估耗时超过 0.1 秒的每个特征模型，完全优化的 ddnnife 变体都显著快于（$p < 0.025$）其他任何变体，唯一例外是不使用核心特征的变体。对于 12 个特征模型，不将核心/死特征用作优化时，运行速度显著快于（$p < 6.5 \cdot 10^{-7}$）完全优化变体。

对于可满足的部分配置，完全优化的变体与不使用核心/死特征的变体表现非常相近，运行时间分别为 4.24 分钟和 4.26 分钟。此外，不使用部分计算和部分遍历的变体紧随其后，运行时间分别为 4.52 分钟和 4.57 分钟。朴素变体在 9 个特征模型上达到超时。对于每个评估时间超过 0.1 秒的特征模型，完全优化的变体都显著快于（$p < 1.5 \cdot 10^{-17}$）朴素变体和不重复利用子树的变体。对于不使用部分计算和部分遍历的变体，在一个和两个特征模型上的运行时间分别没有显著差异。不使用核心/死特征的变体在 7 个特征模型上显著快于（$p < 0.029$）启用了所有优化的 ddnnife。

对于不可满足的部分配置，完全优化的变体总运行时间最短，为 2.40 分钟。省略部分计算和部分遍历时的运行时间

非常相近，分别为 2.51 分钟和 2.60 分钟。不使用基于核心/死特征的优化时，ddnnife 需要 3.85 分钟，比完全优化的变体慢 1.59 倍。此外，对于除两个特征模型之外的所有模型，完全优化的变体都显著更快（$p < 2.0 \cdot 10^{-6}$）。不使用核心/死特征优化时，运行时间的相对增幅远高于可满足配置（慢 1.01 倍）和特征（慢 1.003 倍）。ddnnife 不使用任何优化时，有 9 个特征模型达到 30 分钟的超时时限。

6.4 讨论

研究问题1*d-DNNF* 编译器应用于工业特征模型时的效率如何？我们的结果表明，编译为 d-DNNF 有望用于特征模型计数。90% 的受评估工业特征模型可在 30 分钟内完成编译，84% 甚至可在 10 秒内完成。#SAT 求解器和 d-DNNF 编译器在处理完全相同的特征模型时均能扩展到相同规模。对于那些在一般情况下进行特征模型计数都非常困难的特征模型，编译为 d-DNNF 似乎才会失败。这一观察符合预期，因为常用的 d-DNNF 编译器和 #SAT 的表现非常相似 [28, 37–39]。在大多数环境中，d-DNNF 的内存使用量也不应造成问题，例如，最快的编译器 d4 生成的 d-DNNF 大小最多为 20.1 MB。总的来说，综合考虑 d-DNNF 的大小和编译时间，d4 是最适用于特征模型的 d-DNNF 编译器。

研究问题2哪些工具和工具组合能够计算出正确的基数？countAntom、Ganak、d4、c2d、ddnnife（结合 c2d 和 d4）以及 query-ddnnf 在每一次计算中都得出了相同的基数，我们认为这是这些工具正确性的有力佐证。尤其是，由于有超过 20 万个正确结果，本次提交中提出并在 ddnnife 中实现的算法和优化很可能能够得出正确结果。只有在 dSharp 生成的错误的确定性否定范式上，ddnnife 算出的少数基数有所不同；这很可能是因为 dSharp 在这些情况下提供了错误的 d-DNNF。

sharpSAT 及其衍生工具 dSharp 在许多情况下得出的结果与其余工具不同（sharpSAT：13.4%，dSharp：53.6%）。这些差异可能与 sharpSAT 中由单子句导致的错误有关，其他研究 [72] 和一条 GitHub issue 中也讨论了这一问题。[25] 虽然潜在的修复可能会影响运行时间，但考虑到该错误的性质，我们认为当前测量结果仍可合理反映正确行为下的性能。[72]此外，这两种工具的表现与其他纳入比较的工具相近，因此我们预计修复错误后，结论不会发生实质性变化。我们认为纳入这两种求解器进行比较非常重要，因为它们都曾应用于产品线领域。[25, 52, 63, 69, 73]因此，尽管存在该错误，我们仍在接下来的研究问题中纳入 sharpSAT 和 dSharp。

研究问题3与重复*#SAT* 调用相比，重复使用 *d-DNNF* 来计算特征和部分配置的基数有多高效？对于特征模型计数的重复查询，重复使用 d-DNNF 的速度显著快于对每个考察的特征模型都使用最先进水平的 #SAT 求解器。分析所有 55 个成功评估的特征模型时，使用最快的基于 d-DNNF 的方法总共需要 6.56 分钟，而最快的 #SAT 求解器则需要 2.81天（慢 619 倍）。即使将编译时间计入，ddnnife 与 d4 配合使用也比最快的 #SAT 求解器快至 8,316 倍。对两种查询类型，甚至可能对后续分析重复使用 d-DNNF，还能进一步缩短 d-DNNF 的运行时间。总之，我们认为，将特征模型编译为 d-DNNF 值得投入；与当前最先进水平的重复 #SAT 查询相比，这种方法效率高得多。此前，尚无任何领域对用于重复计数查询的 d-DNNF 进行实证评估，因此我们的结果首次揭示了其性能。显著且大幅的

[25] https://github.com/marcthurley/sharpSAT/issues/5

相较于重复调用 #SAT 求解器所取得的改进表明，重复使用 d-DNNF 在其他领域也可能很有前景。

总体而言，ddnnife 与 d4 的组合比任何其他工具或工具组合都快。对于特征，ddnnife 与 d4 的组合速度最快，处理所有特征模型仅需 10 分钟；其次是 d4-query（40 分钟）以及 ddnnife 与 dSharp 的组合（70 分钟）。对于部分配置，查询数量较少，而且 ddnnife 需要通过部分遍历（参见第 4.2 节）遍历 d-DNNF 中更大的部分，因此其性能优势较小。尽管总体而言，包含 ddnnife 的工具组合在处理部分配置时仍然最快，但对于许多特征模型，尤其是较小的模型，d4-query 更快。不过，d4-query 仍有一个缺点：每次都必须从头开始；而 ddnnife 和 query-ddnnf 可以重复利用 d-DNNF 进行后续分析，也支持处理由多个编译器生成的 d-DNNF。对于特征模型计数的重复查询，采用 d4 作为编译器、ddnnife 作为推理器的组合来应用 d-DNNF，似乎是最有前景的方案。**研究问题 4***ddnnife* 中的优化在运行时间方面能在多大程度上提升效率？所介绍的优化对于提升 ddnnife 的效率至关重要。对于某些特征模型，未进行任何优化的变体会达到 30 分钟的超时限制，而完全优化的变体只需几秒钟。特别是，部分遍历、迭代遍历和重复利用子树都能大幅缩短运行时间。部分计算以及核心/死特征的作用大多较小，但仍能正向影响运行时间。使用核心/死特征进行优化，仅在评估不可满足的部分配置时有显著效果，这与预期相符。对于特征基数，核心和死特征完全不需要遍历 d-DNNF，因为没有任何节点被标记为需要部分遍历。此外，我们针对部分配置使用核心/死特征的方法，只能避免遍历不可满足的配置；因此，在只考虑可满足配置时，其收益会有所降低。

6.5 有效性威胁

将特征模型转换为合取范式每种经过评估的编译器和 #SAT 求解器都使用合取范式作为输入格式。要使用这些工具，必须先将特征模型转换为合取范式。两种等价但语法形式不同的合取范式，可能会导致求解器和编译器的运行时间出现显著差异 [68, 74]。因此，更改合取范式的转换方式可能会影响测得的运行时间。不过，分析多种合取范式转换方式超出了本研究的范围。还需考虑的是，有些合取范式转换虽然保持可满足性，却不能保留满足赋值的数量。这类转换可能导致我们分析中的基数有误。对于以专有格式提供的每个特征模型，我们都使用 FeatureIDE [12] 采用的合取范式转换方式，该方式据推测不会改变满足赋值的数量。其余特征模型（由 Oh 等人 [63]收集）已采用合取范式表示，并曾用于其他实证评估 [63, 68, 75,76]。

求解器/编译器的参数设置所有评估的 #SAT 求解器和编译器都提供可能影响其运行时间的参数。不同的参数可能会显著改变评估工具的性能。不过，我们认为，使用默认参数能够合理地代表实际使用情况。针对每个求解器和编译器进行参数优化不在本文的研究范围内。

随机效应与计算偏差测量用于分析特征模型的求解器运行时间时，由于计算偏差、随机效应或后台进程的影响，不同测量结果之间可能存在差异。为减小这些影响因素的作用，我们对每项测量重复进行了 50 次。此外，在进行测量时，

没有执行其他重要进程。此外，我们采用成熟的统计检验来验证结论的显著性。

多线程工具 在我们的实证评估中，我们决定对所有工具进行单线程评估。由于我们的工具 ddnnife 和 #SAT 求解器 countAntom 支持多线程，启用多线程可能会提升这两种工具的性能。然而，所考虑的问题（即特征基数和部分配置）依赖于多个查询，而这些查询也可以通过其他求解器并行处理。因此，我们认为以单线程方式评估所有工具更为合理。此外，初步实验表明，启用多线程时，这两种工具的性能差异很小。

实验设置的外部效度 如果采用不同的实验设置（例如，查询数量较少），结果以及 d-DNNF 与 #SAT 之间的比较可能会有所不同。不过，所评估的查询是由文献和我们的行业项目所启发的。例如，同质性等分析按定义需要计算所有特征的基数。此外，即使只处理一小部分查询，基于 d-DNNF 的方法通常也比反复调用 #SAT 求解器更快。

工具的外部效度 评估工具的测量性能不一定能推广到其他工具。不过，据我们所知，我们评估了所有公开可用的 d-DNNF 编译器和推理器。对于 #SAT 求解器，我们根据以往实验 [25] 选择了特征模型计数表现最好的两个 #SAT 求解器，以及 2020 年模型计数竞赛的获胜 #SAT 求解器。因此，我们预计结果能够合理地反映这两种方法的性能。

研究对象系统的外部效度 我们的结论不一定适用于其他真实世界特征模型。不过，我们评估了来自多个领域的特征模型，包括操作系统、汽车、金融服务以及其他多个软件系统领域。此外，我们涵盖了广泛的特征数量（11–62,482）、子句数量（1–350,221）、有效配置数量（1,024–10^{1534}），以及求解 #SAT 所需的运行时间（从几毫秒到达到 24 小时超时 [25]）。

7 相关工作

本节讨论以下工作：(a) 使用 d-DNNF 进行模型计数；(b) 在特征模型中使用 d-DNNF；(c) 提出允许进行特征模型计数的替代方法；(d) 在特征模型分析中应用知识编译；或 (e) 介绍其他可高效进行模型计数的知识编译产物。

使用 d-DNNF 进行模型计数 本文考察的每个 d-DNNF 编译器，即 c2d [37],dSharp [38], 和 d4 [39], ，都能够计算输入的满足赋值数量。然而，这些编译器都不支持重复使用给定的 d-DNNF 来计算特征或部分配置的基数。虽然 d4（给定一个合取范式时）和 query-ddnnf 支持计算多个查询，但这些功能尚未在科学出版物中讨论，也未经过评估。总的来说，此前尚无关于重复使用 d-DNNF 进行计数的实证评估。

在特征模型分析中利用 d-DNNF Sharma 等人 [46]使用 d-DNNF 进行均匀随机采样，并开发了工具 *KUS*。在内部，*KUS* 依赖于对给定 d-DNNF 进行计数。虽然可以提取该计数过程，但它不支持应用

假设（此处指包含或排除特征），因此无法用于我们考察的两种查询类型。Bourhis 等人 [77] 将 d-DNNF 用于多种不同的特征模型分析，包括计数。不过，对于计数，他们仅考虑针对整个特征模型的单次计算，而我们关注的是针对特征和部分配置的重复查询。在先前的出版物中，我们 [25] 和 Kübler 等人 [20] 评估了 d-DNNF 编译器在计算有效配置数量这一任务中的表现。不过，在这两篇出版物中，编译器都只是被用作黑盒 #SAT 求解器，实质上是为每次单独计算都编译一个新的 d-DNNF，且不会复用已编译的 d-DNNF。此外，他们没有进行重复查询（例如，查询特征基数）。

特征模型计数的解 在上述研究中，Kübler 等人 [20] 和我们 [25] 计算了工业特征模型的基数。这两项工作都仅限于每个特征模型进行一次查询，并且只将求解器当作黑盒使用；而我们则重用已编译的 d-DNNF，以减少重复查询所需的工作量。

Pohl 等人 [78] 比较了几种求解器类型（即二叉决策图、SAT 和约束满足问题）在不同特征模型分析（包括特征模型计数）中的运行时间。结果表明，在计算有效配置的数量时，二叉决策图的扩展性明显优于 SAT 和约束满足问题。然而，他们的数据集仅包含规模相对较小的特征模型（少于 300 个特征），而我们的基准测试包含数万个特征的特征模型。在先前的工作中，我们发现，编译为二叉决策图仅适用于极少数工业特征模型，且效果明显不如 d-DNNF [43, 44]。

其他研究针对没有 [22] 或只有极少跨树约束的特征模型，提出了用于特征模型计数的临时解决方案 [19]。这些方案在没有或只有极少跨树约束的特征模型上表现良好，但近期研究表明，工业特征模型往往包含极大量的跨树约束（例如，我们的数据集中最多有 15k 个） [25, 44, 68, 79]。

均匀随机采样 一些均匀随机采样方法在内部依赖计数 [46, 63, 69, 80]。例如，$Smarch$ [68] 通过反复调用 sharpSAT 来构建均匀样本。$Spur$ [80]对 sharpSAT 进行改造，将计算节点的值缓存起来，以加快有效配置的推导。不过，与KUS（参见特征模型分析中的 ***d-DNNF*** 利用）一样，无法对$Spur$应用假设。因此，如果不进行重大的概念性改造，也无法计算特征或部分配置的基数。

特征模型分析中的知识编译 二叉决策图（BDD）常用于特征模型分析 [14, 25, 81–84], ，其中一些方案提出使用二叉决策图进行特征模型计数 [25, 44, 78, 85, 86]。然而，二叉决策图难以扩展到许多工业特征模型 [25, 44, 68, 87]。Krieter 等人 [16, 88]采用模态蕴含图来加速产品配置过程中的决策传播。然而，模态蕴含图不适用于特征模型计数。

使用其他知识编译产物进行模型计数 对于若干知识编译产物，模型计数是可处理的（即相对于产物的大小，其复杂度为多项式时间）。根据 Darwiche 和 Marquis 的知识编译图 [36], ，二叉决策图、d-DNNF 和 MOD（满足平滑性与确定性的析取范式）均支持多项式时间计数。在其他研究中，研究者发现句子决策图 [60] 和仿射决策树 [59]也可高效地进行模型计数。尽管理论上这些格式都支持计数，我们在近期工作中比较了 BDD、SDD、EADT 和 d-DNNF 编译器在特征模型计数任务上的性能 [43] ，发现 d-DNNF 编译器效率最高。

增量求解 增量求解器会针对高度相似的多个公式执行查询，并在这些查询之间保留信息，以加快后续调用 [89]。针对常规 SAT，研究者提出了多种求解器，在许多用例中取得了良好效果 [30, 31,89]。通常，这些求解器是基于 DPLL [90] 或 CDCL [91]改造的流行 SAT 求解器。尽管这类增量 SAT 求解器已应用多年，但 #SAT 领域尚未出现此类改造。

8 结论

许多基于特征模型计数的分析，都需要执行各种相似的计数查询，以计算包含某个特征或符合某个部分配置的有效配置数量。我们首次通过复用 d-DNNF，减少了重复计数查询所需的工作量。评估结果表明，与反复调用 #SAT 求解器这一最先进水平的方法相比，复用 d-DNNF 能大幅缩短运行时间。我们发现，对于所有最先进水平的 #SAT 求解器能够计算出结果的特征模型，编译为 d-DNNF 均可扩展。此外，对于成功完成评估的全部 55 个特征模型，我们的工具 ddnnife 与最快的编译器结合，评估所有基数需要 6.5分钟 ，而最快的 #SAT 求解器则需要运行 2.81 天 。对于每个纳入考察的特征模型，我们的工具 ddnnife 都显著快于使用最快的 #SAT 求解器，即使将编译时间计算在内，也能节省高达 99.99% 的 CPU 时间。若不计编译时间，ddnnife 的速度甚至可比最快的 #SAT 求解器快 39,253 倍。总而言之，当需要执行多个查询时，我们建议在特征模型计数中使用 ddnnife 复用 d-DNNF。

致谢

本研究以 Raab [92] 和 Sundermann [93]的论文为基础。

参考文献

[1] David Benavides、Sergio Segura 和 Antonio Ruiz-Cortés。《自动化分析特征模型：20年回顾——文献综述》。载于：信息系统35.6（2010年），第615–708页。[2] Frank J. van der Linden、Klaus Schmid 和 Eelco Rommes。《软件产品线实践：产品线工程的最佳工业实践》。施普林格，2007年。[3] John McGregor。《测试软件产品线》。载于：软件工程测试技术。施普林格，2010年，第104–140页。[4] Don Batory。《特征模型、文法与命题公式》。载于：国际系统与软件产品线会议论文集（*SPLC*）。施普林格，2005年，第7–20页。[5] Thomas Thüm、Sven Apel、Christian Kästner、Ina Schaefer 和 Gunter Saake。《软件产品线分析策略的分类与综述》。载于：*ACM*计算调查（*CSUR*）47.1（2014年），6:1–6:45。[6] Klaus Pohl、Günter Böckle 和 Frank J. van der Linden。《软件产品线工程：基础、原理与技术》。施普林格，2005年。[7] Claudia Fritsch、Richard Abt 和 Burkhardt Renz。《银行业中特征模型的益处》。载于：国际系统与软件产品线会议论文集（*SPLC*）。ACM，2020年。

[8] Marc Hentze、Tobias Pett、Chico Sundermann、Sebastian Krieter、Thomas Thüm 和 Ina Schaefer。《面向多领域产品线的通用解空间采样》。载于：生成式编程：概念与经验国际会议论文集（*GPCE*）。即将出版。ACM，2022年。[9] Marc Hentze、Chico Sundermann、Thomas Thüm 和 Ina Schaefer。《量化问题空间与解空间之间的可变性失配》。载于：模型驱动工程语言与系统国际会议论文集（*MODELS*）。IEEE，2022年。[10] Alexander Knüppel、Thomas Thüm、Stephan Mennicke、Jens Meinicke 和 Ina Schaefer。《真实世界特征模型与产品线研究之间是否存在失配？》载于：欧洲软件工程会议／软件工程基础会议论文集（*ES-EC/FSE*）。ACM，2017年，第291–302页。[11] David Benavides、Pablo Trinidad 和 Antonio Ruiz-Cortés。《使用约束编程对特征模型进行推理》。载于：软件工程与知识工程国际会议论文集（*SEKE*）。2005年，第677–682页。[12] Jens Meinicke、Thomas Thüm、Reimar Schröter、Fabian Benduhn、Thomas Leich 和 Gunter Saake。《使用 FeatureIDE 掌握软件可变性》。施普林格，2017年。[13] Joshua Sprey、Chico Sundermann、Sebastian Krieter、Michael Nieke、Jacopo Mauro、Thomas Thüm 和 Ina Schaefer。《FeatureIDE 中基于 SMT 的可变性分析》。载于：软件密集型系统可变性建模国际工作会议论文集（*VaMoS*）。ACM，2020年。[14] David Benavides、Sergio Segura、Pablo Trinidad 和 Antonio Ruiz-Cortés。《FAMA：用于特征模型自动化分析的工具框架》。载于：软件密集型系统可变性建模国际研讨会论文集（*VaMoS*）。技术报告2007-01，Lero，2007年，第129–134页。[15] Mathieu Acher、Philippe Collet、Philippe Lahire 和 Robert B. France。《Familiar：用于大规模管理特征模型的领域特定语言》。载于：计算机编程科学（*SCP*）78.6（2013），第657–681页。[16] Sebastian Krieter、Thomas Thüm、Sandro Schulze、Reimar Schröter 和 Gunter Saake。《使用模态蕴含图传播配置决策》。载于：软件工程国际会议论文集（*ICSE*）。ACM，2018年，第898–909页。[17] Chico Sundermann、Michael Nieke、Paul Maximilian Bittner、Tobias Heß、Thomas Thüm 和 Ina Schaefer。《#SAT 求解器在特征模型中的应用》。载于：软件密集型系统可变性建模国际工作会议论文集（*VaMoS*）。ACM，2021年。[18] Ruben Heradio、David Fernández-Amorós、José Antonio Cerrada 和 Ismael Abad。《特征图产品计数及其在软件产品线经济模型中的应用：文献综述》。载于：软件工程与知识工程国际期刊（*IJSEKE*）23.08（2013年），第1177–1204页。[19] David Fernández-Amorós、Ruben Heradio、José Antonio Cerrada 和 Carlos Cerrada。《一种可扩展的扩展特征模型精确模型计数与共性计数方法》。载于：*IEEE*软件工程汇刊（*TSE*）40.9（2014年），第895–910页。[20] Andreas Kübler、Christoph Zengler 和 Wolfgang Küchlin。《产品配置中的模型计数》。载于：组件配置逻辑国际研讨会论文集（*LoCoCo*）。开放出版协会，2010年，第44–53页。[21] Sheng Chen 和 Martin Erwig。《优化产品派生过程》。载于：国际系统与软件产品线会议论文集（*SPLC*）。IEEE，2011年，第35–44页。

[22] Rubén Heradio-Gil、David Fernández-Amorós、José Antonio Cerrada 和 Carlos Cerrada。“支持软件产品线的基于共性的分析”。载于：*IET* 软件5.6（2011年），第496–509页。[23] Krzysztof Czarnecki 和 Andrzej Wąsowski。“特征图与逻辑：往返之旅”。载于：国际系统与软件产品线大会论文集（*SPLC*）。IEEE，2007年，第23–34页。[24] Marcílio Mendonça、Andrzej Wąsowski 和 Krzysztof Czarnecki。“基于 SAT 的特征模型分析很容易”。载于：国际系统与软件产品线大会论文集（*SPLC*）。软件工程研究所，2009年，第231–240页。[25] Chico Sundermann、Thomas Thüm 和 Ina Schaefer。“在工业特征模型上评估 #SAT 求解器”。载于：软件密集型系统可变性建模国际工作会议论文集（*VaMoS*）。ACM，2020年。[26] Leslie G Valiant。“枚举与可靠性问题的复杂度”。载于：《计算 *SIAM* 杂志》8.3（1979年），第410–421页。[27] David S Johnson。“NP完全性专栏：持续更新的指南”。载于：《算法杂志》13.3（1992年），第502–524页。[28] Marc Thurley。“sharpSAT——借助高级分量缓存和隐式BCP进行模型计数”。载于：可满足性测试理论与应用国际会议论文集（*SAT*）。施普林格，2006年，第424–429页。[29] Johannes K. Fichte、Markus Hecher 和 Florim Hamiti。“2020年模型计数竞赛”。载于：*ACM* 实验算法学杂志（*JEA*）26（2021年）。[30] Alexander Nadel、Vadim Ryvchin 和 Ofer Strichman。“最终增量 SAT”。载于：可满足性测试理论与应用国际会议论文集（*SAT*）。施普林格，2014年，第206–218页。[31] Niklas Eén 和 Niklas Sörensson。“通过增量SAT求解进行时序归纳”。载于：理论计算机科学电子札记（*ENTCS*）89.4（2003年），第543–560页。[32] Tobias Pett、Sebastian Krieter、Tobias Runge、Thomas Thüm、Malte Lochau 和 Ina Schaefer。“持续集成中产品线采样的稳定性”。载于：软件密集型系统可变性建模国际工作会议论文集（*VaMoS*）。ACM，2021年。[33] Jackson A. Prado Lima、Willian D. F. Mendonça、Silvia R. Vergilio 和 Wesley K. G. Assunção。“高度可配置软件持续集成中的测试用例基于学习的优先级排序”。载于：国际系统与软件产品线大会论文集（*SPLC*）。ACM，2020年。[34] Ayelet Israeli 和 Dror G. Feitelson。“Linux内核：软件演化的一个案例研究”。载于：《系统与软件杂志》（*JSS*）83.3（2010年），第485–501页。[35] Rafael Lotufo、Steven She、Thorsten Berger、Krzysztof Czarnecki 和 Andrzej Wąsowski。“Linux内核可变性模型的演化”。载于：国际系统与软件产品线大会论文集（*SPLC*）。施普林格，2010年，第136–150页。[36] Adnan Darwiche 和 Pierre Marquis。“知识编译图”。载于：《人工智能研究杂志》（*JAIR*）17.1（2002年），第229–264页。[37] Adnan Darwiche。“确定性可分解否定范式编译器”。载于：人工智能大会论文集（*AAAI*）。AAAI 出版社，2002年，第627–634页。[38] Christian Muise、Sheila McIlraith、J Christopher Beck 和 Eric Hsu。“使用 sharpSAT 快速编译 d-DNNF”。载于：人工智能大会论文集（*AAAI*）。AAAI 出版社，2010年。

[39] Jean-Marie Lagniez 和 Pierre Marquis。“改进的决策 DNNF 编译器”。载于：国际人工智能联合会议论文集（*IJCAI*）。国际人工智能联合会议，2017年，第667–673页。[40] Adnan Darwiche。“可分解否定范式”。载于：《*ACM* 学报》48.4（2001年），第608–647页。[41] Umut Oztok 和 Adnan Darwiche。“将合取范式编译为决策 DNNF”。载于：约束编程原理与实践国际会议论文集（*CP*）。施普林格，2014年，第42–57页。[42] Jinbo Huang 和 Adnan Darwiche。“带轨迹的 DPLL：从 SAT 到知识编译”。载于：国际人工智能联合会议论文集（*IJCAI*）。第5卷。Professional Book Center，2005年，第156–162页。[43] Chico Sundermann、Tobias Heß、Michael Nieke、Paul Maximilian Bittner、Jeffrey M. Young、Thomas Thüm 和 Ina Schaefer。“在工业配置空间上评估最先进的 # SAT 求解器”。载于：《实证软件工程》（*EMSE*）（2023年）。即将出版。[44] Tobias Heß、Chico Sundermann 和 Thomas Thüm。“为当前特征模型构建二元决策图的可扩展性”。载于：国际系统与软件产品线大会论文集（*SPLC*）。ACM，2021年，第131–135页。[45] Adnan Darwiche。“理论模型的易处理计数及其在真值维护和信念修正中的应用”。载于：《应用非经典逻辑》11.1-2（2001年），第11–34页。[46] Shubham Sharma、Rahul Gupta、Subhajit Roy 和 Kuldeep S Meel。“知识编译与均匀采样相遇”。载于：编程、人工智能与推理逻辑国际会议论文集。EasyChair，2018年，第620–636页。[47] Ruben Heradio、David Fernández-Amorós、Christoph Mayr-Dorn 和 Alexander Egyed。“支持可变性模型的统计分析”。载于：软件工程国际会议论文集（*ICSE*）。IEEE，2019年，第843–853页。[48] Ebrahim Bagheri 和 Dragan Gasevic。“使用结构度量评估软件产品线特征模型的可维护性”。载于：《软件质量杂志》（*SQJ*）19.3（2011年），第579–612页。[49] Krzysztof Czarnecki 和 Chang Hwan Peter Kim。“基于基数的特征建模与约束：进展报告”。载于：软件工厂国际研讨会论文集（*SF*）。2005年，第16–20页。[50] Wei Zhang、Haiyan Zhao 和 Hong Mei。“一种基于命题逻辑的特征模型验证方法”。载于：形式化工程方法国际会议论文集（*ICFEM*）。施普林格，2004年，第115–130页。[51] Thomas Thüm、Don Batory 和 Christian Kästner。“关于特征模型编辑的推理”。载于：软件工程国际会议论文集（*ICSE*）。IEEE，2009年，第254–264页。[52] Jia Hui Liang、Vijay Ganesh、Krzysztof Czarnecki 和 Venkatesh Raman。“对大型真实世界特征模型进行基于 SAT 的分析很容易”。载于：国际系统与软件产品线大会论文集（*SPLC*）。施普林格，2015年，第91–100页。[53] Jan Burchard、Tobias Schubert 和 Bernd Becker。“用于并行# SAT 求解的放任式缓存”。载于：可满足性测试理论与应用国际会议论文集（*SAT*）。施普林格，2015年，第46–61页。[54] Shubham Sharma、Subhajit Roy、Mate Soos 和 Kuldeep S Meel。“GANAK：一种可扩展的概率精确模型计数器”。载于：国际人工智能联合会议论文集（*IJCAI*）。第19卷。AAAI 出版社，2019年，第1169–1176页。

[55] Adnan Darwiche。《将合取范式编译为可分解否定范式的新进展》。载于：欧洲人工智能会议论文集。IOS Press，2004年，第318–322页。[56] Armin Biere。《PicoSAT 要点》。载于：可满足性、布尔建模与计算期刊4（2008年），第75–97页。[57] Roberto J. Bayardo Jr. 和 Joseph Daniel Pehoushek。《使用连通分量进行模型计数》。载于：人工智能会议（*AAAI*）论文集。AAAI 出版社，2000年，第157–162页。[58] Vladimir Klebanov、Norbert Manthey 和 Christian Muise。《基于 SAT 的程序信息流分析与量化》。载于：定量评估系统国际会议（*QEST*）论文集。施普林格，2013，第177–192页。[59] Frédéric Koriche、Jean-Marie Lagniez、Pierre Marquis 和 Samuel Thomas。《用于模型计数的知识编译：仿射决策树》。载于：国际人工智能联合会议（*IJCAI*）论文集。AAAI 出版社，2013。[60] Umut Oztok 和 Adnan Darwiche。《一种用于句子决策图的自顶向下编译器》。载于：国际人工智能联合会议（*IJCAI*）论文集。AAAI 出版社，2015年，第3141–3148页。[61] Tian Sang、Fahiem Bacchus、Paul Beame、Henry A. Kautz 和 Toniann Pitassi。《结合分量缓存与子句学习以实现高效模型计数》。载于：可满足性测试理论与应用国际会议（*SAT*）论文集。施普林格，2004年，第20–28页。[62] Carlos Ansótegui、María Luisa Bonet 和 Jordi Levy。《工业级 SAT 实例的结构》。载于：约束编程原理与实践国际会议（*CP*）论文集。施普林格，2009年，第127–141页。[63] Jeho Oh、Paul Gazzillo、Don Batory、Marijn Heule 和 Margaret Myers。面向真实世界软件产品线的可扩展均匀采样。技术报告 TR-20-01。德克萨斯大学奥斯汀分校计算机科学系，2020年。[64] Tobias Pett、Sebastian Krieter、Thomas Thüm、Malte Lochau 和 Ina Schaefer。《AutoSMP：采样算法评估平台》。载于：系统与软件产品线国际会议（*SPLC*）论文集。ACM，2021年，第41–44页。[65] Michael Nieke、Jacopo Mauro、Christoph Seidl、Thomas Thüm、Ingrid Chieh Yu 和 Felix Franzke。《特征模型演化的异常分析》。载于：生成式编程：概念与经验国际会议（*GPCE*）论文集。ACM，2018年，第188–201页。[66] Tobias Pett、Thomas Thüm、Tobias Runge、Sebastian Krieter、Malte Lochau 和 Ina Schaefer。《产品线采样：可扩展性挑战》。载于：系统与软件产品线国际会议（*SPLC*）论文集。ACM，2019年，第78–83页。[67] Matthias Kowal、Sofia Ananieva 和 Thomas Thüm。《解释特征模型中的异常》。载于：生成式编程：概念与经验国际会议（*GPCE*）论文集。ACM，2016年，第132–143页。[68] Jeho Oh、Paul Gazzillo、Don Batory、Marijn Heule 和 Maggie Myers。从 *Kconfig* 特征模型进行均匀采样。技术报告 TR-19-02。德克萨斯大学奥斯汀分校计算机科学系，2019年。[69] Daniel-Jesus Munoz、Jeho Oh、Mónica Pinto、Lidia Fuentes 和 Don Batory。《对具有数值特征的特征模型进行产品配置均匀随机采样》。载于：系统与软件产品线国际会议（*SPLC*）论文集。ACM，2019年，第289–301页。[70] Patrick E McKnight 和 Julius Najab。《曼-惠特尼 U 检验》。载于：科西尼心理学百科全书（2010年），第1–1页。

[71] Gail M Sullivan 和 Richard Feinn。《使用效应量——或为何 P值并非全部》。载于：研究生医学教育杂志4.3（2012年），第279–282页。[72] Johannes K. Fichte、Markus Hecher 和 Valentin Roland。《命题模型计数的证明》。载于：第*25*届可满足性测试理论与应用国际会议（*SAT 2022*）。Kuldeep S. Meel 和 Ofer Strichman 编。第236卷。莱布尼茨国际信息学论文集（LIPIcs）。达格施图尔城堡，2022，第30:1–30:24页。ISBN：978-3-95977-242-6。网址：https://drops.dagstuhl.de/opus/volltexte/2022/ 16704。[73] Jeho Oh、Paul Gazzillo 和 Don Batory。《通过均匀采样实现 t向覆盖》。载于：系统与软件产品线国际会议（*SPLC*）论文集。ACM，2019年，第84–87页。[74] Will Klieber 和 Gihwon Kwon。《从 n 个对象中选取 1 个的高效合取范式编码》。载于：形式化验证约束国际研讨会（*CFV*）论文集。施普林格，2007年。[75] Christian Kaltenecker、Alexander Grebhahn、Norbert Siegmund、Jianmei Guo 和 Sven Apel。《软件配置空间的基于距离的采样》。载于：软件工程国际会议（*ICSE*）论文集。IEEE，2019年，第1084–1094页。[76] Quentin Plazar、Mathieu Acher、Gilles Perrouin、Xavier Devroey 和 Maxime Cordy。《可配置系统的 SAT 解均匀采样：我们已经做到这一步了吗？》。载于：软件测试、验证与确认国际会议（*ICST*）论文集。IEEE，2019年，第240–251页。[77] Pierre Bourhis、Laurence Duchien、Jérémie Dusart、Emmanuel Lonca、Pierre Marquis 和 Clément Quinton。特征模型推理：基于编译的方法与直接方法。2023年。[78] Richard Pohl、Kim Lauenroth 和 Klaus Pohl。《当代特征模型自动化分析操作算法方法的性能比较》。载于：自动化软件工程国际会议（*ASE*）论文集。IEEE，2011年，第313–322页。[79] Alexander Knüppel、Thomas Thüm、Stephan Mennicke、Jens Meinicke 和 Ina Schaefer。《真实世界特征模型与产品线研究之间是否存在不匹配？》。载于：软件工程（*SE*）论文集。Matthias Tichy、Eric Bodden、Marco Kuhrmann、Stefan Wagner 和 Jan-Philipp Steghöfer 编。德国信息学会，2018年，第53–54页。[80] Dimitris Achlioptas、Zayd S Hammoudeh 和 Panos Theodoropoulos。《满足赋值的快速完美均匀采样》。载于：可满足性测试理论与应用国际会议（*SAT*）论文集。施普林格，2018年，第135–147页。[81] Mathieu Acher、Philippe Collet、Philippe Lahire 和 Robert B. France。《特征模型切片》。载于：自动化软件工程国际会议（*ASE*）论文集。IEEE，2011年，第424–427页。[82] Mathieu Acher、Patrick Heymans、Philippe Collet、Clément Quinton、Philippe Lahire 和 Philippe Merle。《特征模型差异》。载于：高级信息系统工程国际会议（*CAiSE*）论文集。施普林格，2012年，第629–645页。[83] Marcílio Mendonça、Andrzej Wąsowski、Krzysztof Czarnecki 和 Donald Cowan。《大规模特征模型的高效编译技术》。载于：生成式编程与构件工程国际会议（*GPCE*）论文集。ACM，2008年，第13–22页。[84] Jeho Oh、Don Batory、Margaret Myers 和 Norbert Siegmund。通过随机采样寻找高性能产品线配置。技术报告。德克萨斯大学奥斯汀分校，2016年。

[85] Martin Fagereng Johansen、Øystein Haugen、Franck Fleurey、Erik Carlson、Jan Endresen 与 Tormod Wien。《面向产品线的敏捷自动化交互测试技术》。载于：国际软件与系统测试会议论文集（*ICTSS*）。施普林格，2012年，第39–54页。[86] Marcílio Mendonça、Moises Branco 与 Donald Cowan。《S.P.L.O.T.：软件产品线在线工具》。载于：面向对象编程、系统、语言与应用会议论文集（*OOPSLA*）。ACM，2009年，第761–762页。[87] Thomas Thüm。《Linux 的二叉决策图？可变性的知识编译挑战》。载于：国际系统与软件产品线会议论文集（*SPLC*）。ACM，2020年。[88] Sebastian Krieter、Rahel Arens、Michael Nieke、Chico Sundermann、Tobias Heß、Thomas Thüm 与 Christoph Seidl。《用于演化特征模型的模态蕴含图增量构造》。载于：国际系统与软件产品线会议论文集（*SPLC*）。ACM，2021年，第64–74页。[89] Katalin Fazekas、Armin Biere 与 Christoph Scholl。《SAT 求解中的增量内处理》。载于：可满足性测试理论与应用国际会议论文集（*SAT*）。Mikoláš Janota 与 Inês Lynce 编。施普林格，2019年，第136–154页。[90] Harald Ganzinger、George Hagen、Robert Nieuwenhuis、Albert Oliveras 与 Cesare Tinelli。《DPLL (T)：快速判定过程》。载于：计算机辅助验证国际会议论文集（*CAV*）。施普林格，2004年，第175–188页。[91] Joao Marques-Silva、Inês Lynce 与 Sharad Malik。《冲突驱动子句学习 SAT 求解器》。载于：可满足性手册。IOS Press，2009年，第131–153页。[92] Heiko Raab。《利用 d-DNNF 高效分析基于基数的特征模型》。学士学位论文。乌尔姆大学，2022。[93] Chico Sundermann。《#SAT 求解器在产品线中的应用》。硕士学位论文。布伦瑞克工业大学，2020年。