arXiv:2302.06351v1 [cs.DS] 13 Feb 2023

对称性检测预处理器的工程设计

Markus Anders[1],Pascal Schweitzer[1],and Julian Stieß[2]

[1]达姆施塔特工业大学[2]凯撒斯劳
滕-兰道莱茵兰-普法尔茨工业大学

2023年2月14日

摘要

用于组合对象对称性检测的最先进求解器正逐渐发展为日益复杂的软件库。大多数求解器最初是针对组合学问题设计的（NAUTY、 BLISS、 TRACES、 DEJAVU）。它们擅长处理输入中复杂的核心部分。另一些则专注于具有稀疏性的实际实例，擅长处理输入中相对简单但规模极大的子结构（SAUCY）。在实践中，这些差异体现为它们在不同类型图类上的性能存在显著差异。

我们为对称性检测设计了一种预处理器。该工具旨在缩减输入图中稀疏的大型子结构。对于大多数实际实例，该预处理器显著缩短了许多最先进求解器的整体运行时间。同时，我们的基准测试表明，额外开销可以忽略不计。

总体而言，我们获得了在所有基准图上都具备竞争力的单一算法。因此，预处理器弥合了专注于组合图的求解器与面向大型实际图的求解器之间的差距。事实上，在大多数实际实例上，组合方案的表现显著优于以往的最先进水平。

1 引言

在 SAT [20, 5, 13], SMT [12], QBF [21], CSP [15], ILP [25, 27, 17] 等众多算法应用领域中，利用对称性是不可或缺的手段。然而，要利用对称性，我们必须先计算出它们。

许多类型的对象都可以高效地建模为图，因此这些对象的对称性对应于图的对称性。这包括公式、方程组、有限关系结构等（见 [29]）。因此，计算这些对象的对称性可归结为计算图的对称性。我们将计算图的对称性的过程称为对称性检测。

最先进水平的对称性检测工具包括 NAUTY [26]、 SAUCY [11]、 BLISS [19]、 TRACES [26]和 DEJAVU [7]。它们以顶点着色图作为输入，并输出该图的所有对称性。所有最先进水平的工具都基于所谓的个体化-细化（IR）范式（第 3.3 节）。然而，它们在所采用的搜索策略、剪枝

本研究得到欧洲研究委员会（ERC）通过欧盟地平线2020研究与创新计划（EngageS：资助编号820148）的支持。

不变量、对称性处理以及各种其他启发式方法方面存在显著差异（见 [26, 7]）。这也反映在它们针对不同图类的性能差异上。

我们想重点介绍两个求解器性能差异显著的例子，即“实际图”和“组合图”。对于大型实际图，例如源自 SAT、QBF、混合整数规划或道路网络的图，求解器 SAUCY 的表现显著优于所有其他求解器（例如，参见 [5] 中的结果，或本文第 9 节的基准测试）。事实上， SAUCY 是针对可满足性检查设计的，专门为这类图进行了精心优化。直观来说，源自实际应用的图往往规模较大，但结构相对简单。另一方面，对于几乎所有相对于规模而言较难处理的图类（例如射影平面、CFI图及其他正则组合对象）， TRACES和 DEJAVU 凭借更为复杂的搜索策略，轻松胜过其他求解器（更细致的讨论请参见 [7] 和 [26] ）。图 1 展示了在一个组合学中的困难图类和一个典型的大型实际图类上， SAUCY 与 DE-JAVU 之间的巨大差异。

当然，只提供针对特定类型图而设计的求解器并不是理想的情况。例如，这意味着我们必须选择一个求解器，因此需要了解所面对的输入类型。此外，对于由不同类型图组合而成的输入，我们也会难以应对。因此，自然而然地，我们更希望有一个能在所有图上都表现出色的单一求解器。

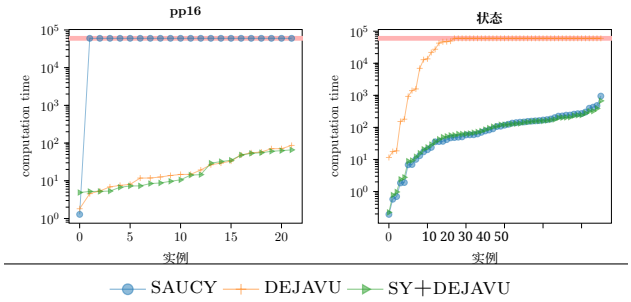


图 1：比较求解器在一组难解的组合图（**pp16**，左）和大型实际图（**状态**，右）上的表现。超时时间为 $60s$ （红色条形）。 SY+DEJAVU 指的是采用了本文所述预处理器的 DEJAVU 。

一种常用范式是添加预处理器，使求解器能够更广泛地应用于计算问题。预处理器的使用确实已在 SAT、QBF 或最大可满足性领域 [14, 10, 24]带来了无数成功案例。在这些应用中，如今通常会对所有输入进行预处理。相比之下，迄今还没有用于对称性检测的预处理器。事实上，McKay 和 Piperno [26] 明确指出，在他们看来，使用其工具之前，“特定类型 [的图] 应当通过预处理来处理”。

除了提升性能之外，使用预处理器还有其他多种好处。首先，可以独立于主求解器的设计来解决图的初始简化问题。这一点尤其可取，因为最先进水平的对称性检测求解器实现十分复杂，而且其内部工作机制的详细描述大多难以获得。其次，如果某些情况能在运行求解器之前得到可靠处理，预处理器甚至可以降低求解器实现的复杂度。最后，在通用预处理器中实现的策略可以同时供所有求解器使用。

鉴于目前缺乏用于对称性检测的预处理器， TRACES 等工具会使用复杂的子程序，在计算前（有时也在计算过程中）简化某些低度顶点（参见实现 [2]）。总的来说，问题在于能否设计出一种通用预处理器，既能简化输入，又能惠及所有最先进的求解器。

贡献。 我们实现了首个用于对称性检测的预处理器 SASSY 。该预处理器在设计上兼容所有处于最先进水平的对称性检测工具。我们的基准测试（第 9 节）证实，在许多实际图类上，使用该预处理器的求解器配置显著优于最先进水平的

方法。同时，即使对于预处理技术不起作用的图，该预处理器也只会引入可忽略不计的开销。

预处理器弥合了专注于困难组合图（Traces、 dejavu、 bliss、 nauty）的求解器与专注于大型实际图（saucy）的求解器之间的差距。借助预处理器，前一类求解器如今在大多数实际图上的表现优于 saucy。此外，在大多数数据集上， saucy 本身也因预处理器而加速。

技术。 预处理器主要实现用于处理输入和输出均较稀疏的图（例如实际图）的技术。具体而言，它由以下几个构建模块组成，我们将在论文中逐一讨论：

1. 一个用于对输入图应用约简技术的框架。具体而言，该框架涵盖了如何从约简图中将对称性重建回输入图，这既包括理论层面，也包括实际操作层面（第4节）。

2. 一种高效移除度为 0 和 1 的顶点的技术（第 5.1 节和第 5.2 节）。3. 在避免引入带颜色的边或有向边的前提下，部分移除度为 2 的顶点（第 5.3 节）。4. 一种用于探测“稀疏自同构”的基于 IR 的探测技术（第6节）。5. 利用商图的概念，发掘连通分量和齐次连接（第 7 节）。

虽然 sassy 是首个用于对称性检测的通用预处理器，但我们想指出，(2) 的一种形式已在 Traces 中实现。除 (2) 之外的所有技术都是新的贡献。不过，我们确实想提及，(4) 和 (5) 从求解器的现有技术中汲取了一些灵感。我们将在相应章节中详细说明这一点。

2 预处理器的设计理念

设计预处理器时，主要挑战之一是明确哪些技术和方法属于预处理器的职责范围，哪些任务应由主算法解决。另一个棘手的问题是预处理器与主求解器之间的接口，以及用户/预处理器接口。

在设计预处理器时，我们既遵循了概念原则，也考虑了技术要求。

概念原则。 从概念层面来说，我们的目标是设计高效的预处理子程序，以简化对称性计算任务。自然，预处理器只应采用相对于主算法运行时间而言较快的过程。

我们的预处理器设计以所谓的颜色细化算法为核心。颜色细化是进行对称性检测的一种强大启发式方法。所有最先进的求解器都会持续、反复地应用它。因此，在颜色细化所需时间内或接近该时间内运行的过程，都可以放心使用。

预处理器的总体思路是移除图中已通过颜色细化处理解决的子结构，或将其因式分解。主要难点在于如何尽可能高效地检测并利用这些子结构。从本质上说，任何能够高效处理的部分都应当谨慎处理，并采用恰当的方法。

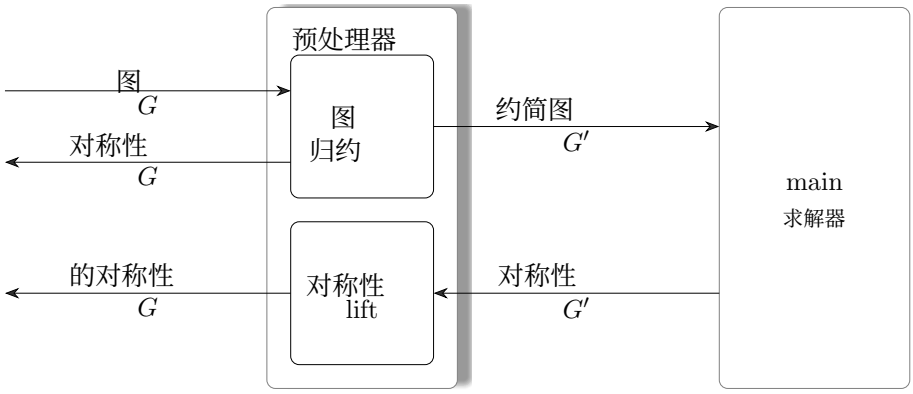


图 2：我们提出的预处理器/主求解器以及用户/预处理器接口。在图约简过程中，预处理器可能已经确定了 G 的部分（或全部）对称性。随后，约简实例将传递给主求解器。

尽管主求解器本身有能力且能够较为高效地处理此类子结构（尤其是能够在多项式时间内完成处理），但若在处理更复杂的部分之前不先处理这些子结构，或未能尽可能高效地处理它们，就会拖慢整个流程。

总体而言，我们需要在子程序的效率、有效性和通用性之间取得平衡 .

技术要求。 在技术层面上，我们希望预处理器能够兼容所有最先进的求解器。因此，我们需要使用一个适用于所有现有工具的通用接口。所有工具都会读取顶点着色图并输出对称性。因此，这也是预处理器所使用的接口。

预处理器读取一个顶点着色图，并输出一个传递给主求解器的约化顶点着色图。此外，预处理器可能已经确定了部分或全部对称性，并立即将其输出给用户。还有一个技术细节：主求解器计算出的约简图的对称性，按定义并不是原图的对称性。为此，预处理器采用反向翻译（即一种后处理形式），将主求解器发现的对称性提升为原始输入图的对称性。

我们的设计如图 2 所示。

3 预备知识

除非另有说明，图 G 均为有限简单无向图。顶点 v的邻域记作 $N(v)$，其度 为 $\deg(v) := |N(v)|$。对于顶点集 $V' \subseteq V(G)$，其邻域 是集合 $N[V'] := (\bigcup_{v\in V'} N(v)) \setminus V'$。

图 G 的着色是一个从顶点到某个颜色集合的映射 $\pi\colon V(G) \to \mathcal{C}$。一个（颜色）类 C 是由同一种颜色的顶点组成的集合 $\pi^{-1}(c)$。当着色 π 是单射时，称其为离散着色。换言之，在离散着色中，每个顶点都有自己独一无二的颜色。除非另有说明，我们研究由顶点集 V、边集 E和着色 π组成的着色图 $G = (V, E, \pi)$。稍微滥用一下记号，将无着色图 $G = (V, E)$中的点对 (G, π) 与 (V, E, π) 等同起来。对于顶点子集 $V' \subseteq V$，图 $G = (V, E, \pi)$ 的诱导子图为 $G[V'] = (V', E', \pi|_{V'})$，其中 $E' = \{e \in E \mid e \subseteq V' \times V'\}$。

双射 $\varphi\colon V \mapsto V$称为自同构（对称性），当且仅当 $(\varphi(V)\ \varphi(E)) = (V,\ E)$

（将 φ 逐一应用于 E边中的顶点）。如果 G 带有颜色， φ 还必须保持颜色（即满足 $\pi(\varphi(v)) = \pi(v)$）。自同构的数量可能随图的规模呈指数增长。在复合运算下，自同构构成一个置换群。包含（带颜色）图 G 所有自同构的自同构群为 Aut(G)。自同构 $\varphi \in$ Aut(G) 的支撑集是 supp(φ) := $\{\varphi(x) \neq x \mid x \in V(G)\}$，即未被该自同构固定的顶点。自同构子集 $S \subseteq$Aut(G) 是 Aut(G) 的生成集，当且仅当对 S 中的置换进行穷尽复合后能够得到 Aut(G) 的所有元素。我们记作 $\langle S \rangle$ =Aut(G)。这使得 Aut(G) 可以得到简洁编码。求解器通常只输出 Aut(G) 的一个生成集。

3.1 颜色细化

颜色细化算法是一种经过深入研究的过程 [8, 9, 26]。它是同构和对称性计算中不可或缺的基础子程序。直观地说，给定一个着色图，它会以特定方式拆分颜色，从而得到一种“细化后”的着色。关键的是，这一过程不会改变图的对称性，而细化后的着色已经限制了可能的解。

形式上，如果对于任意一对（不一定互不相同的）颜色类 C_1、C_2， C_1中的所有顶点在 C_2中都有相同数量的邻居（即，对于所有 v、$v' \in C_1$， $|N(v) \cap C_2| = |N(v') \cap C_2|$ 。），则图的着色 π 称为等划分。给定一种着色 π，颜色细化会计算出一种等划分细化 π'（即一种等划分着色 π'，满足 $\pi'(v) = \pi'(v')$ 蕴含 $\pi(v) = \pi(v')$）。事实上，它计算的是最粗的等划分细化。关键的是， $G = (V, E, \pi)$的自同构也是 $G = (V, E, \pi')$的自同构（反之亦然）。因此，使用 π'而非 π进行处理既有益处，也是常规做法。

颜色细化可以通过一种实现方式，使其最坏情况下的运行时间为 $\Theta((n+m)(\log\ n))$（见 [9]）。从实现角度来看，它是最关键的子程序，因此经过了高度优化。

3.2 商图

对于图 G（原本未着色）的均衡着色 π，商图 $Q(G, \pi)$记录了一个颜色类中的顶点在另一个颜色类中拥有的邻居数量信息。商图是一个完全有向图，其中每个顶点都有一个自环。 $Q(G, \pi)$ 的顶点集为 $V(Q(G, \pi)) := \pi(V(G))$，也就是说，它是 π 下各顶点颜色构成的集合。 $Q(G, \pi)$ 的顶点以其在 G 中所代表的颜色着色。我们将边 (c_1, c_2) 的颜色设为颜色为 c_1 的顶点拥有的颜色为 c_2 的邻居数量（可能为 $c_1 = c_2$）。回顾一下，由于 π 是等划分， c_1 中所有顶点在 c_2中拥有相同数量的邻居。当且仅当两个图相对于最粗等价着色所得的商图相等时，它们才无法通过颜色细化区分（例如参见 [8]）。

3.3 个体化细化算法

个体化细化框架是一种通用框架，用于计算同构、自同构和规范标号（见 [26]）。这些算法通常在一种特殊的树上运行，即所谓的*IR*树。下面简要介绍 IR 树的构造方式。要理解本文内容，并不需要了解构造过程中的许多细节。只有第6节所述的技术依赖于构造过程的某些部分。更详尽的介绍请见 [26, 7]。

树的每个节点 x 都对应于输入图的一个均衡着色 π_x 。叶节点对应于离散着色。IR树最重要的性质是，它们

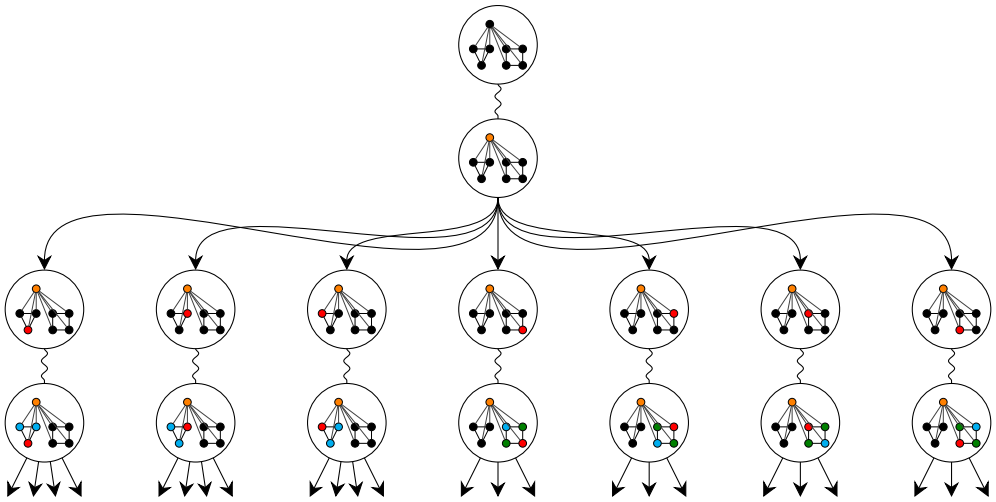


图3：IR树的构建。由波浪线连接的节点对，在IR树中对应一个节点。一对节点中位于上方的节点（根节点除外）展示了对一个顶点进行个体化的过程，而位于下方的节点则展示了对上方节点应用颜色细化后得到的最终着色。

具有同构不变性，这意味着在 G 和 φ（其中 φ 是一个同构）上，我们得到同构的IR树。

设 $(G,\ \pi)$ 为输入图。设 π' 是 π 的最粗均衡细化。我们令 IR树的根节点对应于 π'。

在 IR 树的每个节点 x 中，都会从对应的着色 π_x 中选取一个非平凡颜色类（即，必须以同构不变的方式选取一个 $C = \pi_x^{-1}(c)$，其中 $|C| > 1, C$ ）。如果不存在非平凡颜色类，那么 x 就是叶节点，其对应的着色是离散的。否则，对于每个 $v \in C$，我们将 x_v 定义为 IR 树中 x 的子节点。令 π_{x_v} 表示与 x_v对应的着色。我们可以按如下方式从 π_x 得到 π_{x_v} 。从 π_x 开始，我们先人为地单独标出v（即，个体化 v）。这意味着我们设置 $\pi_{x_v}(v) := c'$，其中 $c' \notin \pi(V(G))$（同样，c' 以同构不变的方式选取）。然后，我们使用颜色细化来细化着色，从而得到均衡着色 π_{x_v}。图3展示了如何从一个图构建 IR 树的示意图。

我们可以从 IR 树推导出自同构。如果 $\pi_1,\ \pi_2$ 是树的叶节点，即离散着色，那么 $\varphi := \pi_1^{-1} \circ \pi_2$ 定义了 $V(G)$ 上的一个置换。虽然 φ 不一定是自同构，但我们可以高效地检验它是否为自同构（通过检查 $\varphi(G) = G$）。利用这种方法，可以计算出 Aut(G) 的全部元素。从根本上说，这是因为以这种方式比较所有叶节点对，就能得到 G 的所有自同构（或者说，在应用自同构剪枝时，得到 Aut(G) 的一个生成集；见 [26]）。

实际实现当然会采用许多策略来提高搜索效率，例如自同构剪枝、不变量，或可能省略 IR 树部分内容的搜索策略。关于 IR 树和实用技术的更全面介绍，请见 [26, 7]。

4 用于化简图的工具箱

现在，我们开始介绍用于简化图以检测对称性的技术。我们的目标始终是高效地减少图中的顶点和边的数量。

不过，我们需要警惕一些技术细节。每当我们修改图时，都必须确保不会丢失任何对称性，或者立即输出那些将会丢失的对称性。此外，我们还必须确保预处理完成后，约简图的对称性能够映射回原图的对称性。毕竟，我们关注的是原图的对称性。为了简化这一过程，我们首先介绍一些贯穿整篇论文所使用的一般技术。

我们所描述的第一类技术会将顶点集为 V 的输入图 G 修改为顶点集为 $V' \subseteq V$ 的另一个图 G'，使得

1. $\mathrm{Aut}(G)|_{V'} \subseteq \mathrm{Aut}(G')$ and （对称性保持）

2. $\mathrm{Aut}(G)|_{V'} \supseteq \mathrm{Aut}(G')$。 （对称性提升）

这里，$\mathrm{Aut}(G)|_{V'}$ 指的是将每个 $\varphi \in \mathrm{Aut}(G)$ 的定义域限制为 V'（值域限制为 $\varphi(V')$）后得到的映射集合。如果条件 (1) 和 (2) 成立，那么 V' 也必定在 $\mathrm{Aut}(G)$ 下保持不变。

在这些条件下，限制到 V' 上会得到一个自然的同态 $p\colon \mathrm{Aut}(G) \to \mathrm{Aut}(G')$。于是，轨道-稳定子定理（见 [18, 定理 2.16]）意味着，如果 $S' \subseteq \mathrm{Aut}(G)$ 是 $\mathrm{Aut}(G')$ 的生成集 S 的一组提升，即 $p(S') = S$，那么 $\mathrm{Aut}(G) = \langle S', \ker(p) \rangle$（其中 $\langle \Gamma \rangle$ 表示由 Γ 生成的群，见 [30]）。这里，$\ker(p) = \{\varphi \in \mathrm{Aut}(G) \mid p(\varphi) \neq 1\}$ 是 p 的核，而 1 表示单位元。

总体而言，这使我们能够将 $\mathrm{Aut}(G)$ 的计算分解为两部分：计算图中被移除部分的自同构，以及约简图的自同构。

对于这些技术而言，关键在于能够根据 G 高效计算 G' 以及 $\ker(p)$ 的一个生成集，并且能够根据 $\mathrm{Aut}(G')$ 的一个生成集高效计算提升集合 S'。

特别是，我们需要一种高效的后处理技术，将自同构提升到已约简的部分。为此，我们引入两种贯穿预处理器设计全过程的工具。

规范表示字符串。 在预处理期间，我们从原图中移除的部分可能与图的其他部分对称（即处于同一轨道中）。因此，在计算出约简图的对称性之后，我们需要将约简图的对称性提升为原图的对称性。具体而言，提升后的对称性必须正确映射所有被移除的部分。为了简化对称性的提升，我们引入与剩余顶点相关联的表示字符串。这些字符串编码了被移除顶点的性质（即“同构类型”）。编码存储在一个合适的剩余顶点的颜色中。如果一个剩余顶点随后被映射到另一个顶点，那么由这些字符串表示的相应子图就会以规范方式彼此映射。

我们通过以下方式正式定义这一过程：表示映射 $\mathcal{R}(v)\colon V \mapsto V^*$ 从顶点映射到顶点序列。假设图 $G := (V, E, \pi)$ 被约化为 $G' := (V', E', \pi')$，其中 $V' \subseteq V$ 且 $E' \subseteq E$。我们要求满足以下条件：

1. 对所有 $v \in V'$，均有 $\mathcal{R}(v) := vS$，且 $S \in V^*$；也就是说，每个剩余顶点必须首先表示自身。

2. 对于所有 $v \in V \setminus V'$，$\mathcal{R}(v) := \epsilon$ 成立，即已删除顶点不代表任何顶点。

3. 对于每个已删除顶点 $v \in V \setminus V'$，至多存在一个 $v' \in V'$ 和至多一个 $i \in \mathbb{N}$，使得 $v := \mathcal{R}(v')_i$，即每个已删除顶点至多由一个剩余顶点代表一次。

对于剩余图 $\varphi \in \mathsf{Aut}(G')$ 的每个自同构，我们现在定义其提升双射 $\varphi_{\mathcal{R}}(v) \in \mathsf{Sym}(V)$（即 V 上的对称群）。我们定义 $\varphi_{\mathcal{R}}(v) :=$

$$\begin{cases} \varphi(v) & \text{if } v \in V' \\ \mathcal{R}(\varphi(v'))_i & \text{if } v = \mathcal{R}(v')_i \text{ for } v' \in V', i \in \mathbb{N} \\ v & \text{if } v' \neq \mathcal{R}(v')_i \text{ for all } v' \in V', i \in \mathbb{N}. \end{cases}$$

We 调用 $\mathcal{R}$ 一个 规范表示映射 如果 $\varphi_{\mathcal{R}} \in \mathrm{Aut}(G)$ 对所有 $\varphi \in \mathrm{Aut}(G')$.

请注意，根据定义，规范表示映射可以串联使用，也就是说，如果我们多次化简一个图 G，只需依次反向应用相应的规范表示映射，直到得到 G 的一个自同构。我们甚至可以通过组合这些函数，将串联的规范表示映射改写为一个单一映射。（更准确地说，我们必须通过拼接，将字符串的字符串解释为普通字符串。）

稀疏自同构与恢复。 以下各节中我们隐含使用的一个概念是自同构的稀疏编码。一种常见做法是采用置换的轮换表示法，即只存储每个非不动元素及其像 [18]。不过，具体采用何种编码并不重要。关键是，自同构的编码所占空间应与其支撑集的大小成正比，即为 $\mathcal{O}(|\mathsf{supp}(\varphi)|)$。这意味着，稀疏自同构，也就是几乎固定所有点的自同构，可以用简洁的方式编码。

利用规范表示映射 $\mathcal{R}$ 和稀疏自同构编码，可以高效地将约简图 G' 的自同构提升为原图 G 的自同构。事实上，只需用其表示字符串替换顶点，便可在与提升的支撑集大小成线性的时间（和空间）内计算出提升。

事实 1 给定 $\varphi \in \mathrm{Aut}(G')$，提升 $\varphi_{\mathcal{R}} \in \mathrm{Aut}(G)$ 可在时间 $\mathcal{O}(|\mathrm{supp}(\varphi_{\mathcal{R}})|)$ 内计算得出。

我们指出，规范表示通常实际上能确保提升后的支撑集尽可能小。如果表示映射 $\mathcal{R}$ 尊重核的轨道，则称其具有这样的性质：当 v_1 和 v_2 属于 $\ker(p)$ 的同一轨道时，$v_1 \in \mathcal{R}(v) \Leftrightarrow v_2 \in \mathcal{R}(v)$。

事实 2 如果 $\mathcal{R}$ 尊重核的轨道，那么 $p(\psi) = \varphi$ 意味着 $|\mathrm{supp}(\varphi_{\mathcal{R}})| \leq |\mathrm{supp}(\psi)|$。

我们随后介绍的所有表示都遵循核的轨道。

我们应当指出，除了 SAUCY 之外，所有最先进的求解器都没有为稀疏自同构提供接口，即无法在时间 $\mathcal{O}(|\mathsf{supp}(\varphi)|)$ 内访问一个自同构的接口。相反，访问只能在 $\Omega(|V|)$ 时间内完成。如果用户应用程序正确使用稀疏自同构接口，那么对于包含大量稀疏自同构的图（许多实际图都属于这种情况），这可以显著缩短运行时间。大多数求解器无论如何都会在内部产生 $\Omega(|V|)$ 的开销来处理自同构，从而使稀疏接口变得没有必要。由于我们的预处理器并非如此，并且为了确保用户应用程序能够获得潜在的运行时间收益，预处理器找到的自同构当然可以通过稀疏方式访问。

5 移除低度顶点

我们介绍的第一类高效约简技术会移除度较低的顶点。我们针对度为 0、1 和 2（类似地为 $n-1$、$n-2$）的顶点提出了相应策略。

我们想指出， Traces [2]的实现中可以找到对度为 0 和 1 的顶点进行预处理的技术，尽管 [26] 中的描述采用了不同的技术，并未明确提及这些技术。 Traces 对度为 0、1 的实现与我们提出的策略不同之处在于，它不会 在移除度为 0 和 1 的顶点之前执行颜色细化，因此面临不同的挑战，也需要采用不同的技术。 Traces还包含在其主要计算过程中 处理度为 2 的顶点的代码，我们将在下文进一步讨论。

5.1 度为0的顶点

对度为 0 的顶点（以及类似地，对 $n-1$）进行预处理很简单。该算法会检测由度为 0 的顶点组成的颜色类。我们令 V' 为度大于 0 的顶点集合。只需移除度为 0 的顶点，并且完全不在 $\mathcal{R}$ 中表示它们， $\mathcal{R}$ 便确实定义了一个规范表示映射。

限制 p 到 V' 的核 ker(p) 按如下方式计算。对于 G 中每个由度为 0 的顶点组成的颜色类，我们输出该类上对称群的生成元。

5.2 度 1顶点

穷举地移除所有度为 1 的顶点（类似地，也移除 $n-2$）实质上会从图中移除所有树状附属部分。众所周知，对这些树状附属部分应用颜色细化会产生轨道划分——但有一个显著例外：无法确定这些附属部分的根节点是否属于同一轨道。

我们可以递归地移除度为 1 的顶点。设 G 是一个包含度为 1 的顶点的图。我们描述 G' 和 $\mathcal{R}$，其中移除一个度为 1 的颜色类。

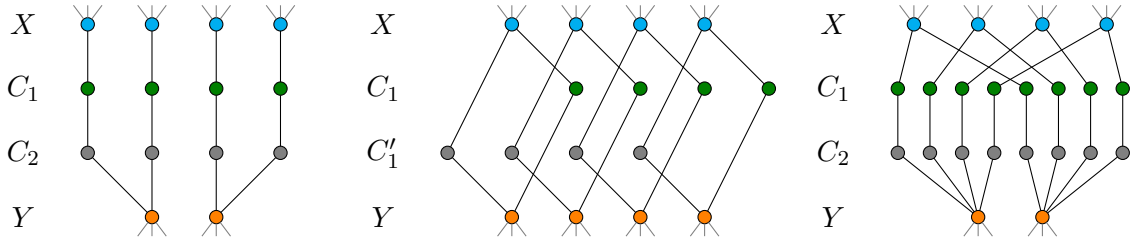
设 C 表示这样一个度为 1 的颜色类。由于该着色是均衡着色， C 中顶点的所有邻居都属于同一个颜色类 P。当 $P=C$ 时，我们得到大小为 2 的连通分量。此情形可类似于度为 0 的顶点的归约来处理，因此我们假设 $P \neq C$。我们将 C 划分为类 C_1、.....、C_m，其中 $c \in C_i$ 与 $p_i \in P$ 相邻。对于表示映射，我们设置 $\mathcal{R}(p_i) := p_i C_i$ （其中 C_i 可以按任意顺序出现）。我们设置 $G' := G \setminus \{C\}$。着色 π 保持不变。注意，对于 G'， π 仍然是一个均衡着色。核 ker(p) 是对每个 $i \in \{1,\ldots,m\}$ 的对称群 Sym(C_i) 的直积（ C 之外的点保持不变）。随后可以重复这一过程，直到所有度为 1 的顶点都被移除。

根据构造，该归约保持对称性且可进行对称性提升，因此 Aut(G) =$\langle S'$,ker(p)$\rangle$。与之前一样， S 是 Aut(G') 的一个生成集，而 S' 是相应的提升集。

5.3 度为 2 的顶点

如果允许预处理器生成的图包含有向带颜色的边，就有一种简单的归约可以移除所有度为 2 的顶点：我们可以将两个顶点 v_1 和 v_2 之间、满足 deg(v_i) ≥ 3 且所有内部顶点的度均为 2 的路径多重集，编码为 v_1 和 v_2 之间的一条有向带颜色的边（另见 [22, 引理 15] 的证明）。

不过，这种方法也有缺点：首先，有些求解器并未实现有向边和带颜色的边。其次，即使求解器实现了这类边，使用有向边和带颜色的边也会带来额外开销 [28]。直观来说，虽然移除



(a) 唯一端点。 (b) 混淆匹配。 (c) 混淆边翻转。

图 4：可约化的度为 2 的模式。

所有度为 2 的顶点肯定能大幅缩小规模，但被移除路径的一部分复杂性只是转移到了边的颜色编码中。因此，我们要求细化过程考虑边颜色。这使作为核心子程序的颜色细化变得更加复杂。

基于这些原因，如果可能，我们倾向于以不需要引入有向边或带颜色的边的方式移除度为 2 的顶点。通常，我们不清楚如何高效地将带颜色顶点构成的路径替换为无色边。我们想指出，TRACES 的技术不会移除度为 2 的顶点，而是使用专用代码 [26] 处理仅置换度为 2 的顶点的自同构。特别是，这意味着在 IR 搜索期间进行颜色细化计算时，度为 2 的顶点仍会产生开销。

具有唯一端点的非分支路径。我们介绍一种启发式方法，发现它常常适用于实际数据集。该方法将内部顶点度为 2、连接两个颜色类的路径编码为一组直接连接端点的边。不过，只有当这些路径能够根据边集被无歧义地重建时，才会这样做。尤其是，插入的边不得与现有边冲突。

我们检测长度为 t 、连接不同颜色类 X 和 Y 且其内部顶点的度为 2 的路径。在 X 的每个顶点中，恰好应有一条这样的路径从该顶点开始（见图 4a）。更正式地，设 $X = C_0, C_1, \dots, C_t, C_{t+1} = Y$ 是满足以下条件的颜色：

- X 中的顶点没有邻居属于 Y，
- C_i 中的 $i \in \{1, \dots, t\}$ 顶点的度为 2，
- C_i 中的 $i \in \{1, \dots, t\}$ 顶点在 C_{i-1} 和 C_{i+1}中各有一个邻居，并且
- X 中的每个节点在 C_1中恰好有一个邻居

然后我们通过 $V' := V - (C_1 \cup \dots \cup C_t)$ 和 $E' := E(G[V']) \cup E''$ 定义 $G' = (V', E')$，其中 E''由满足以下条件的有序对 (x, y) 组成：存在一条路径 $(x, c_1, \dots, c_t, y)$，且 $c_i \in C_i$。相应的表示映射为 $\mathcal{R}(x) = x c_1 c_2 \cdots c_t$，其中 $(x, c_1, \dots, c_t, y)$ 是从 x 到某个满足 $c_i \in C_i$ 的顶点 $y \in Y$ 的唯一路径。

注意，新引入的边 E'' 在 X 和 Y之间构成一个双正则二分图，其中 X 中的顶点的度为 1。不难验证，这会产生一个尊重核轨道的规范表示映射。

混淆匹配。对于 $|X| = |Y|$这一特定情形，预处理器有专门的快速代码。在这种情况下，E' 编码了 X 和 Y之间的一个完美匹配。

该技术的一个小扩展会检查 C_i 的其他选择是否也满足所需属性，并产生完全相同的匹配 E''。事实上，如果在 X 和 Y 之间还存在一个通过颜色类 C'_1、.....、$C'_{t'}$ 进行的、编码了 E'' 的另一个匹配，我们也会删除 C'_i 中的顶点

（见图4b）。专用代码使用数组，可以高效地检查匹配是否一致。

我们应该提到，在实现中，我们只对长度为$t = 1$的路径执行检查。事实证明，$t = 1$ 这一特殊情况很常见；事实上，多个这样的路径编码同一个匹配的情况也很常见，尤其是在混合整数规划和 SAT 基准测试中。

混淆边翻转。另一种也能高效处理、且以往技术未涵盖的情况是，X 和 Y 之间由 $|X||Y|$ 条颜色相同且唯一的路径相连。在这种情况下，每个顶点 $x \in X$ 都通过一条路径与所有 $y \in Y$ 相连（见图4c）。很容易看出，删除所有此类路径既保持对称性，又能实现对称性提升（这与第7.1节所述的边翻转有关）。

形式化地说，假设 $X = C_0$，C_1、.....、C_t，$C_{t+1} = Y$是颜色，且满足

- 对于 C_i 中的 $i \in \{1, \ldots, t\}$ 个顶点，其度为2，
- 对于 C_i 中的 $i \in \{1, \ldots, t\}$ 个顶点，它们在 C_{i-1} 和 C_{i+1} 中都有一个邻居，并且
- X 中的每个节点在 C_1 中恰好有 $|Y|$ 个邻居，对应的路径都以所有$y \in Y$结尾。

该技术进而从图中移除所有 C_0，C_1、.....、C_t。

现在我们来考虑如何计算这种归约的提升。遗憾的是，规范表示字符串不足以表达这种提升：我们需要确定 C_0，C_1、.....、C_t 的映射方式，而这取决于这两者：X 和 Y 中的顶点。我们不能简单地将 C_0，C_1、.....、C_t附加到其中一个颜色类的规范表示字符串上。不过，如果我们知道 X 和Y的映射方式，就可以轻松地还原原始对称性：假设某个对称性将 $x \in X$映射到 x'，并将 $y \in Y$ 映射到 y'。这仅仅意味着，在提升过程中，我们需要将连接 x 和 y 的路径映射到连接 x' 和 y' 的路径。因此，仍然可以非常轻松、高效地计算提升。

在实现中，我们确实会将顶点 C_0、C_1、.....、C_t同时写入 X 和 Y 的表示字符串中，这违反了不得出现重复条目的形式要求。然而，我们使用一种编码技巧来表示这些重复条目，从而在对称重构期间触发特殊代码。

6 探测稀疏自同构

现在，我们提出一种探测稀疏自同构的策略。该策略受到 SAUCY 提出的一种启发式方法的启发：SAUCY （部分也包括 TRACES）实现的启发式方法，能够快速检测并利用“稀疏自同构”。从本质上说，SAUCY 不仅能够利用输入中的稀疏性（例如图中的低度顶点），还能够利用输出中的稀疏性（例如支撑集较小的自同构）[11]。

虽然预处理器已经利用上述低度顶点技术来利用稀疏性，但在本节中，我们将讨论一种更通用的技术，它不依赖于图的特定子结构。具体而言，我们提出一种针对稀疏自同构的探测策略。如果探测成功并发现了自同构，我们就将它们“除去”，通过个体化打破对称性。

SAUCY 中实现的策略会持续检查两个着色 π_1，π_2 判断交换大小为 1 的颜色类中的顶点（即单元素集顶点）并固定所有其他顶点，是否能得到图的一个自同构。更正式地说，我们定义置换 $\varphi_{\pi_1,\ \pi_2}(v) :=$

$$\begin{cases} v & \text{if } |\pi_1^{-1}(\pi_1(v))| \neq 1 \vee |\pi_2^{-1}(\pi_2(v))| \neq 1 \\ \pi_2^{-1}(\pi_1(v)) & \text{otherwise.} \end{cases}$$

然后，我们只需检查 $\varphi_{\pi_1,\ \pi_2}$ 是否确实是 G的一个自同构。该检查可以在时间 $\mathcal{O}(\sum_{v\in \mathrm{supp}(\varphi_{\pi_1},\pi_2)} 1 + \deg(v))$ 内完成。为此，我们只需检查 $\varphi_{\pi_1,\ \pi_2}$ 在 $\varphi_{\pi_1,\ \pi_2}$ 的支撑集及其邻域上是否诱导出一个自同构。

SAUCY 会在 IR 树的深度优先搜索过程中检查局部自同构。随后，它可以存储自同构的相关信息，并在内部利用自同构的存在。然而，出于预处理的目的，我们希望让图变得更简单或更小。具体来说，我们希望能够立即消去已发现的自同构。这一过程应独立于后续所采用的策略类型。因此，我们提出一种专为预处理设计的新搜索策略。

我们的探测策略只搜索可直接用于缩减图的自同构。思路如下：对于要缩减的颜色类，我们尝试收集一些自同构，使它们能够传递地置换整个颜色类中的所有顶点。这样即可确认该颜色类是一个轨道。随后，我们便可以将该颜色类中的任意顶点个体化。相比之下，如果我们只有一些自同构，而它们合起来也无法在该颜色类上实现传递作用，那么就不清楚该如何以有利的方式处理图。具体来说，由于某些顶点可能不属于同一轨道，我们无法确定应该将哪个顶点个体化。

6.1 L-颜色类上的有界 IR 探测

现在我们来介绍有界 IR 探测算法。从理念上讲，该算法结合了 TRACES和 DEJAVU 所使用的随机路径探测，以及 SAUCY所使用的稀疏自同构检测。

（算法 *1* 的描述。）（伪代码见算法 1。）该算法的输入为一个着色图 $G = (V, E, \pi)$、一个颜色类 $C_{probe} = \pi^{-1}(c)$以及长度上界 L。它输出一组自同构 Φ 和一个细化 π的着色 π'。如果探测未成功，则 $\Phi = \{\}$ 且 $\pi' = \pi$。否则，$\langle\Phi\rangle$ 在 C_{probe} 上具有传递性，而 π' 是通过个别化一个顶点并进行细化，从 π 得到的。

我们计算任意 IR路径（即 IR树中的一条根路径），起始于对 C_{probe}中的一个顶点进行个别化。路径的计算长度不超过 L。

最初，算法从两个不同的顶点 $v_1,\ v_2 \in C_{probe}$ 出发，同时检查其中两条路径。每次个体化后，它都会检查自同构 $\varphi_{\pi_1,\ \pi_2}$（如上所定义，即映射对应的单元素集）是否为自同构。如果在完成例如 L' 次个体化后确实如此，我们就将所有后续路径限制为 L'。

之后，对于每个顶点 $w \in C_{probe} \setminus \{v_1, v_2\}$，我们计算一条以对 w 进行个体化为起点的 IR路径。我们希望找到一个将 v_1 映射到 w 的自同构。如果我们针对每个 w 都发现了一个自同构，就返回自同构集合 Φ，在 π 中将 v_1 个体化，进行细化以获得 π'，然后返回 Φ 和 π'。

（算法 *1* 的正确性。）算法的正确性直接源于我们验证了所有自同构这一事实。也就是说，任何被声称为自同构的映射确实都是自同构。由于我们对每个自同构都进行了验证，这就证明了 C_{probe} 是 $\mathrm{Aut}(G,\ \pi)$ 的一个轨道。由于我们返回了构造该轨道所需的全部自同构（即返回了 Φ），根据轨道-稳定子定理（见 [18]），我们有 $\langle\Phi \cup \mathrm{Aut}(G,\ \pi')\rangle = \mathrm{Aut}(G,\ \pi)$。

（算法 *1* 的实现。）我们想进一步说明该算法的实现。事实上，尽管它可以得到非常高效的实现，但通常必须谨慎使用。总的来说，我们需要决定何时以及多频繁地采用探测策略，同时还要决定使用哪个深度界 L。接下来将介绍三种受限的探测策略，每种策略都有专用代码。

算法 1： 在颜色类 C_{probe} 中进行有界 IR 探测，最多沿长度为 L 的路径。

函数BoundedProbeIR(G, π, C_{probe}, L)
  输入： 图 $G=(V,E,\pi)$，其中 π 是等划分，π 的颜色类为 C_{probe}，长度上界 L
  输出：（等划分）着色 π'，自同构集合 Φ
  $\Phi := \{\}$；// 自同构群集合
  选择顶点 v_1，$v_2 \in \pi^{-1}(C_{probe})$;
  对于 $i \in \{1,2\}$ **执行**
    $\pi_i := \pi$;
    将 v_i 个体化到 π_i中;
    颜色细化(G，π_i);
  $L_C := [C_{probe}]$; // 颜色类列表
  当 $|L_C| < L$ **时执行**
    如果 $\varphi_{\pi_1,\ \pi_2}(G,\ \pi) = (G,\ \pi)$ **则**
      跳出; // 找到自同构
    $C := \pi_1$ 的非平凡颜色类;
    L_C += $[C]$；// 将 C 追加到 L_C
    在 π_1 中个体化一些 $v \in \pi_1^{-1}$（C）；
    颜色细化（G，π_1）；
    在 π_2 中个体化一些 $v \in \pi_2^{-1}$（C）；
    颜色细化(G，π_2);
  如果 $\varphi_{\pi_1,\pi_2}(G,\pi) \neq (G,\pi)$ **则**
    返回 π, $\emptyset$; // 探测失败
  else
    $\Phi := \Phi \cup \{\varphi\}$;
  对于 $w \in C_{probe} \setminus \{v_1, v_2\}$ **执行**
    将 π_2 重置为 π；// 本质上 $\pi_2 := \pi$
    将 w 在 π_2中个体化;
    颜色细化(G，π_2);
    对 $C \in L_C$ **执行**
      在 π_2 中将一些 $v \in \pi_2^{-1}(C)$ 个体化;
      颜色细化(G，π_2);
    如果 $\varphi_{\pi_1,\pi_2}(G,\pi) \neq (G,\pi)$ **则**
      返回 π, $\emptyset$; // 探测失败
    else
      $\Phi := \Phi \cup \{\varphi\}$;
  个体化 v_1 在 π；// 成功；在 (G,π) 中个体化 v_1
  返回颜色细化(G，π)，Φ;

6.2 探测的高层策略

我们将说明如何在预处理器中应用 IR 探测。

1-IR 探测，任意类大小。第一个变体对图中的每个颜色类进行长度上限为 $L = 1$ 的探测。为了提高效率，实现还会进行额外的轨道计算：如果我们已经知道 v_1 和 w 在迄今发现的自同构下属于同一轨道，那么就无需为 w 计算路径。轨道计算

使用并查集的一种变体来执行。

∞-IR 探测，颜色类大小 2。 第二种变体是对大小为 2 的单个颜色类进行无长度限制的探测。但如果探测已在 $L = 1$中成功，我们就设置 $\pi = \pi_1$，并选取一个新的大小为 2 的颜色类继续。

对于这个特定情况，算法确实可以使用更少的辅助数据结构，而且无需进行上文所述的额外轨道计算。

∞-IR 探测，颜色类大小 B。 最后一种变体是在无长度限制的情况下，对大小不超过 B 的一个颜色类进行探测。本质上，我们采用与颜色类大小为 2 时相同的做法。不过，这里会用到更多辅助结构以及轨道计算。因此，最终的过程开销显著更大，在下文介绍的高层策略中也就不那么受青睐。在实现中，我们使用 $B = 8$。

7 利用商图

现 介绍另一组利用商图 $Q(G, \pi$的技术).

7.1 边翻转与平凡分量的移除

首先，我们介绍如何高效地在颜色类之间翻转边。设 C_1, C_2 是 π的两个不同颜色类。假设它们之间有 m 条边。 C_1和 C_2 之间的边数最多为 $|C_1||C_2|$。如果 $|C_1||C_2| - m < m$，我们可以将每条边翻转为非边，并将每条非边翻转为边，从而减少图中的边总数。由于这一操作在同构下保持不变且可逆，因此图的自同构群不会改变。

反复且彻底地执行边翻转后，大小为 1 的颜色类中的顶点（单点顶点）将变为度为 0 的顶点。事实上，我们可以直接移除单点，而不必执行涉及单点的边翻转，且不会改变自同构群。

我们要说明的是，在实现中，我们使用一个规范表示映射来跟踪所有已移除的顶点，其中也包括已移除的单点。因此，论文中介绍的所有技术都使用字符串表示。除了用作全局规范表示映射外，我们还允许对顶点进行重命名，从而在剩余 n 个顶点时，将所有剩余顶点映射到区间 $\{1, 2, \ldots, n\}$。

7.2 连通分量

一种比移除单点更通用的策略，是利用商图的连通分量。

考虑图 G 关于顶点着色 π 的商图 $Q = Q(G, \pi)$。 Q 的（弱）连通分量将 G 的顶点集划分为若干连通性均一的部分。这使我们能够独立处理各个分量：

引理 1 若 $D_1, \ldots, D_t$ 是商图 $Q(G, \pi)$ 的连通分量，则有 自同构群$(G, \pi) = \Pi_{i=1}^{t}$ 自同构群$((G, \pi)[D_i])$。

通过翻转两个颜色类之间的边，我们只能缩小 $Q(G, \pi)$ 的连通分量。因此，先穷尽地翻转边，再考察连通分量是有益的（另见 [23]）。

此前，这类连通分量已用于同构和自同构测试 [16, 19]。（在这些情境中，并不使用边翻转，而是通过非齐次连接来刻画商图中的边，两者等价。）

关于实现，我们不显式构造商图，而是计算商图的连通分量。我们首先对所有完全连接的颜色类执行边翻转，也就是说，当 C_1，C_2 之间的边数等于 $|C_1||C_2|$ 时执行边翻转。接着，我们对用于计算连通分量的基础算法进行如下修改：通常，算法会确定顶点 v 的邻域 $N(v)$，并将该邻域加入 v 所属的连通分量。我们的修改是在加入 $N(v)$ 的同时，也加入 $\pi^{-1}\pi(v)$（即加入整个颜色类）。由此，算法便能将顶点划分为商图的各个连通分量。

我们使用这种方法，分别对商图的每个连通分量执行算法 1 的各个变体。

我们想指出，预处理完成后，理论上也可以利用商图的各个连通分量，针对由这些分量诱导的子图分别调用主求解器。这样一来，子图会更小，处理过程也可以并行化。不过，在我们的测试中，预处理完成后通常只剩下一个连通分量，或者有一个很大的连通分量和若干较小的连通分量。因此，至少到目前为止，我们发现分别调用求解器并无益处。

8 技术调度

现在，我们将介绍预处理器何时以及如何组合前几节所述的技术。

预处理器的第一步是应用颜色细化，以生成均衡着色。在整个算法过程中，着色始终保持均衡；必要时（即应用探测技术时）会重新进行颜色细化。此外，我们的实现允许用户自由指定各种技术的调度。

用于生成基准测试的调度如下。首先，我们移除单点。然后，移除度为 0 和 1 的顶点，并对度为 2 的顶点应用前文所述的启发式方法。接着，我们翻转边，并利用商图分量，应用针对稀疏自同构的不同探测变体。之后再次移除单点。最后，只要图中仍包含度为 0 或 1 的顶点，且图的顶点数至少减少了 25%，我们就重复这一调度。请注意，这保证了相对于原图的规模，该调度最多只会重复对数次。

该实现名为 SASSY。它使用 C++ 实现，并采用了 DEJAVU 的颜色细化（该实现本身融合了 TRACES 和 SAUCY 中的颜色细化实现）。该实现为开源软件，可在 [3] 免费获取。

9 基准测试

我们将基准测试部分分为三个主要部分：首先，我们检验在预处理技术据称有效的图类上，应用预处理器是否能加快最先进求解器的速度。同时，我们还检验在这些技术无效的图上，是否引入了过多开销。其次，我们在广泛的实际数据集上，将使用该预处理器的求解器配置与最先进的 SAUCY 和 TRACES 进行性能比较。基准测试的最后一部分见附录 B；我们将在其中分析预处理器所用各项技术的单独影响，考察这些技术合用时确实产生影响的一些图。

下文中，每当我们先应用预处理器，再执行主 SOLVER 时，就用 SY+SOLVER 表示这一组合配置。所有基准测试均在同一台机器上运行

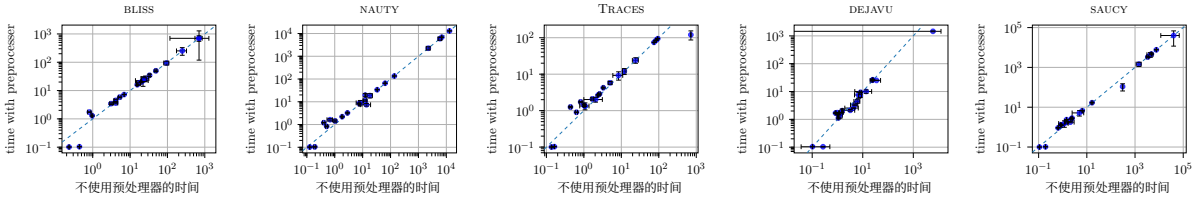


图5：使用 SASSY 与不使用 SASSY 的求解器在**portfolio comb (p)**上的对比。

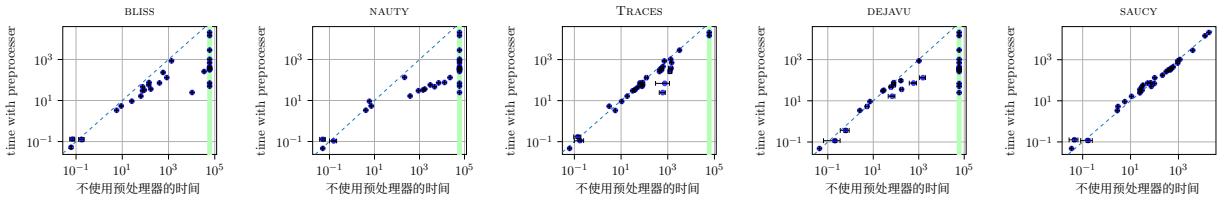


图6：使用 SASSY 与不使用 SASSY 的求解器在 **portfolio pract (p)** 上的对比。超时为 $60s$。绿色条形表示未使用预处理器时超时的实例。启用预处理器后，没有实例发生超时。

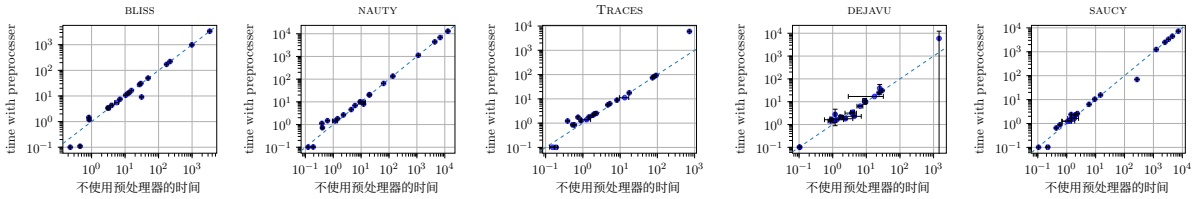


图7：使用 SASSY 与不使用 SASSY 的求解器，在**portfolio comb** 上（未置换）。

该机器配备英特尔酷睿 i7 9700K、64GB 内存，运行 Ubuntu 20.04。我们使用了 NAUTY/TRACES 2.6、SAUCY 3.0、 BLISS 0.73 和 DEJAVU 1.2（ DEJAVU 的误差界限为 1%，使用 4 个线程）。所有基准测试均连续运行 3 次。我们报告平均值和标准差。

传统上，测试对称性检测求解器的默认方法是先随机重排所有给定的基准图 [26, 7]。然而，我们认为，对于许多实际图而言，这是否是测试工具的正确方法并不明确：初始顺序往往并非任意的，甚至可能确实编码了信息。此外，在实际图上，求解器通常能在多项式时间内运行：缓存效率以及高效处理简单情况等方面，可能比剪枝技术等更为重要。因此，对未经重排的实例进行基准测试，或许对实践者更有参考价值。

无论如何，我们以两种方式进行了所有基准测试：首先，按照传统方式随机重排实例；其次，使用实例未经改动的初始顺序。我们用 **(p)** 标记随机重排过的集合。

9.1 预处理与未预处理

我们准备了两组实例，用于分别测试对各个求解器应用预处理器的影响。

portfolio_pract: 本组旨在衡量：对于具有大量可利用结构的图，预处理对给定求解器是否值得采用。因此，本组包含实际图。请注意，我们会在下一节中对实际图进行更全面的测试。为弥补

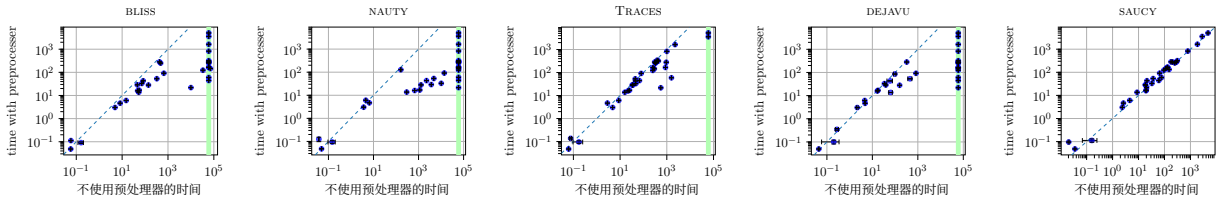


图8：使用 SASSY 的求解器与不使用 SASSY 的求解器在 **portfoliopract** 上的对比（未进行置换）。超时设为 $60s$。绿色条形表示未使用预处理器时超时的实例。启用预处理器后，没有实例超时。

portfolio pract，我们从所有 SAUCY 基准测试集中选取了规模最大的 5 个实例（如果有），并从源自计算任务（混合整数规划和 SAT）的测试集中均匀随机选取了 5 个实例（当然，所有求解器使用的都是相同的实例）。

结果汇总于图6和图8。我们得出结论，对于 BLISS、 NAUTY 和DEJAVU ，预处理器在大多数实例上都显著提升了性能。对于 TRACES，性能也有所提升，尤其是超时情况减少了。不过，提升幅度没有那么明显。对于 SAUCY，则没有明显改善。请注意，对于大多数实例，标准差太小，无法在图中显示。

portfolio comb：这组测试旨在衡量：在没有或几乎没有可利用结构的图上应用预处理器所产生的开销（也就是说，预期预处理器不会产生效果）。为此，我们从组合学中选取了种类繁多的图；求解器通常会在这些图上进行评测 [26]。我们选取的子集几乎涵盖了基准测试库 [2]中的每个图类，其中包含一个图（cfi、grid、grid-sw、had、had-sw、超立方体、kef、latin、latin-sw、格、mz、paley、pp、ran10、ransq、sts、sts-sw、ranreg、tran、triang 和收缩多足图）。在适用的情况下，我们选取了约有1000个顶点的图：请注意，由于组合图通常规模越大就越难处理，因此这里对图的规模作了限制。若求解器难以快速完成某个实例，我们就选取更小的图，或完全省略相应的数据集。请注意，由于我们要衡量预处理开销，因此只关注求解器能在合理时间内完成的实例。如果求解器一开始就需要很长时间才能求解某个实例，那么预处理器的开销总是微不足道的。请注意，这些限制**仅适用于 portfolio comb**：本文测试的所有其他数据集均不限制实例规模，也未手动挑选实例。

结果汇总于图5和图7中。请再次注意，对于许多实例而言，标准差低到无法在图中显示。总体而言，对于测试的求解器，我们得出结论：预处理器的开销可以忽略不计。不过，有两个格外引人注目的实例：首先，对于 DEJAVU，有一个实例的标准差非常高。这个实例是一个克罗内克眼翻转图，而众所周知， DEJAVU 在处理它时会遇到困难 [6]。其次，对于 TRACES，有一个耗时特别长的异常实例。我们在附录A中详细分析并讨论了该实例。在那里，我们得出结论：运行时间增加是由与 TRACES 使用的一种启发式方法发生了意料之外的相互作用所致。事实上，我们可以构造一个在结构上等同于预处理后实例的图，其运行速度比原始实例快6倍。

9.2 与最先进水平的比较

用于求解大型实际图的最先进求解器是 SAUCY。因此，我们将所有使用预处理器的求解器与不使用预处理器的 SAUCY 进行比较。由于 TRACES 也包含低次技术以及稀疏自同构检测的一些特性，我们还比较了性能

	最先进水平		本文				
图类	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
DAC（p）	0.51 ± 0.057	2.49 ± 0.072	**0.38** ± 0.006	0.91 ± 0.016	0.5 ± 0.002	0.41 ± 0.058	0.46 ± 0.005
状态（p）	7.32 ± 0.031	12.55 ± 0.281	**6.79** ± 0.075	6.8 ± 0.069	6.8 ± 0.062	6.83 ± 0.085	6.8 ± 0.074
互联网（p）	0.19 ± 0.005	2.47 ± 0.379	**0.14** ± 0.003	0.15 ± 0.004	**0.14** ± 0.002	**0.14** ± 0.004	**0.14** ± 0.003
ISPD（p）	7.12 ± 0.069	7.04 ± 0.085	5.48 ± 0.046	**5.43** ± 0.023	5.46 ± 0.005	5.45 ± 0.025	5.45 ± 0.008
MIP2017（p）	22.3 ± 0.22	803.63 ± 10.469	**14.07** ± 0.171	92.76 ± 2.796	28.59 ± 0.234	15.52 ± 0.324	26.54 ± 0.128
SAT2021（p）	2217.61 ± 0.627	3645.33 ± 11.963	**1701.62** ± 10.411	1856.39 ± 10.138	1939.54 ± 4.926	1786.68 ± 6.005	1763.12 ± 6.214
SAT2021-up（p）	1886.87 ± 4.472	2948.5 ± 25.254	**1439.5** ± 2.972	1538.71 ± 8.639	1650.68 ± 3.577	1508.91 ± 2.327	1481.28 ± 3.556

图 9：在各种大型实用图（**随机置换**）集合上的基准测试结果，超时限为 60s。内存耗尽也计为超时。基准测试比较了使用预处理器（“SY+”）的求解器配置与先进水平 SAUCY 和 TRACES。所示数值为集合中所有实例的总秒数。采用连续运行 3 次的平均值和标准差。粗体条目表示相应集合中运行时间最快的配置。

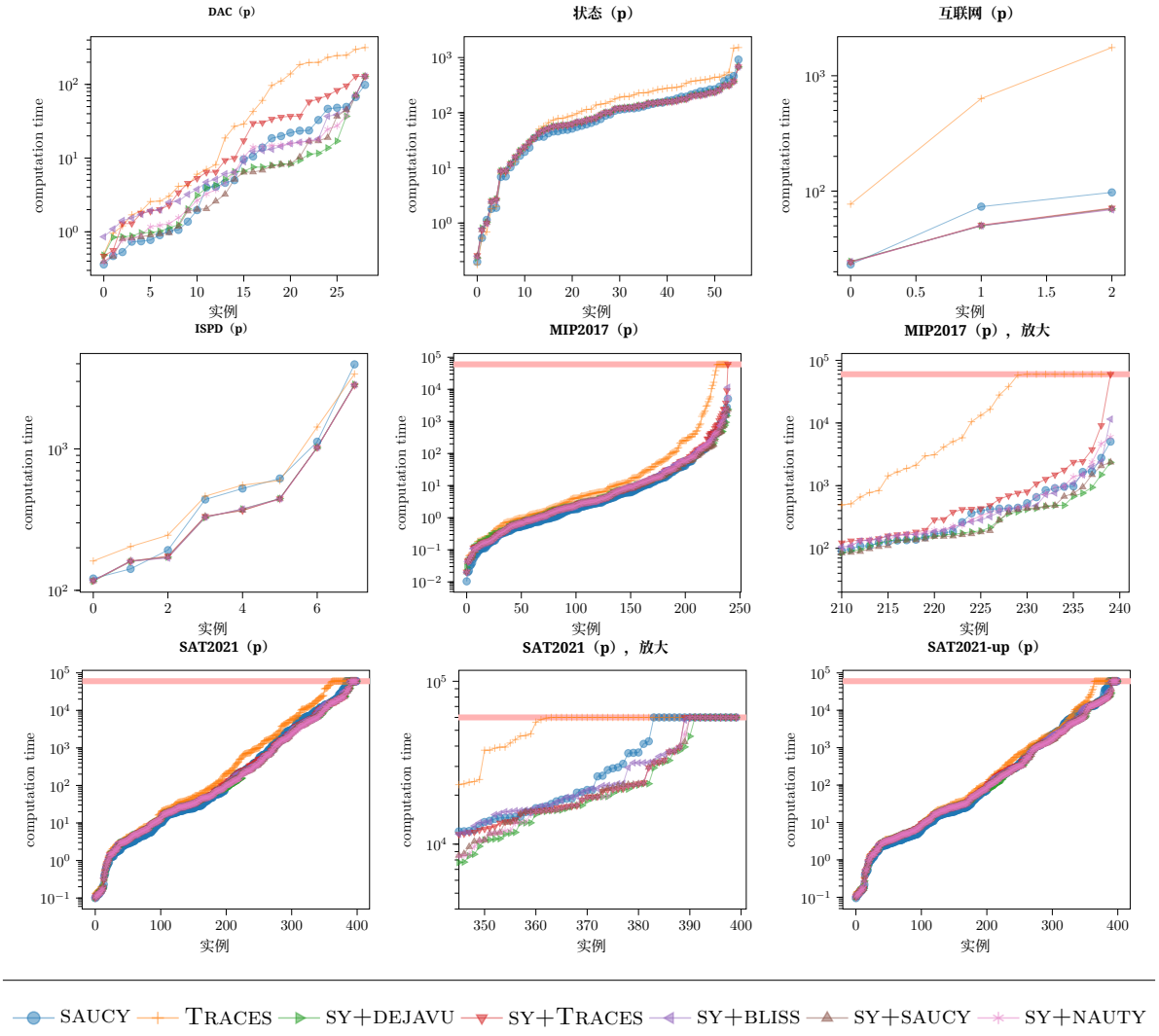


图 10：图 9 中各集合的详细图表。红色条形表示超时。实例按运行时间排序。

	最先进水平		本文				
图类	SAUCY	TRACES	SY+DEJAVU	SY+TRACES	SY+BLISS	SY+SAUCY	SY+NAUTY
dac	0.35 ± 0.008	2.47 ± 0.005	0.28 ± 0.001	0.81 ± 0.002	0.37 ± 0.002	**0.27** ± 0.002	0.34 ± 0.001
状态	**2.89** ± 0.054	7.58 ± 0.158	3.85 ± 0.048	3.85 ± 0.04	3.85 ± 0.041	3.85 ± 0.038	3.85 ± 0.043
互联网	0.15 ± 0.002	2.23 ± 0.022	**0.13** ± 0.000	**0.13** ± 0.001	**0.13** ± 0.001	**0.13** ± 0.000	**0.13** ± 0.000
ispd	3.83 ± 0.028	4.84 ± 0.059	3.7 ± 0.057	3.7 ± 0.061	3.7 ± 0.057	3.7 ± 0.054	**3.68** ± 0.039
MIP2017	10.96 ± 0.158	774.09 ± 0.578	**9.12** ± 0.165	84.42 ± 0.201	21.46 ± 0.331	10.66 ± 0.109	21.04 ± 0.163
SAT2021	1292.76 ± 1.641	2855.57 ± 10.636	**881.69** ± 0.982	1058.97 ± 3.283	1149.04 ± 3.73	990.96 ± 3.323	988.15 ± 2.662
SAT2021-up	1144.06 ± 3.648	2393.85 ± 4.799	**780.02** ± 6.009	903.93 ± 2.517	1027.61 ± 3.815	876.77 ± 2.535	865.38 ± 2.578

图 11：在各种大型实用图（**未置换**）上的基准测试结果，超时时间为 60s。内存耗尽也计为超时。基准测试比较了使用预处理器（“SY+”）的求解器配置与先进水平 SAUCY 和 TRACES。所示数值为该组中所有实例的总秒数。采用连续运行 3 次的平均值和标准差。粗体条目表示给定集合中最快的运行时间。

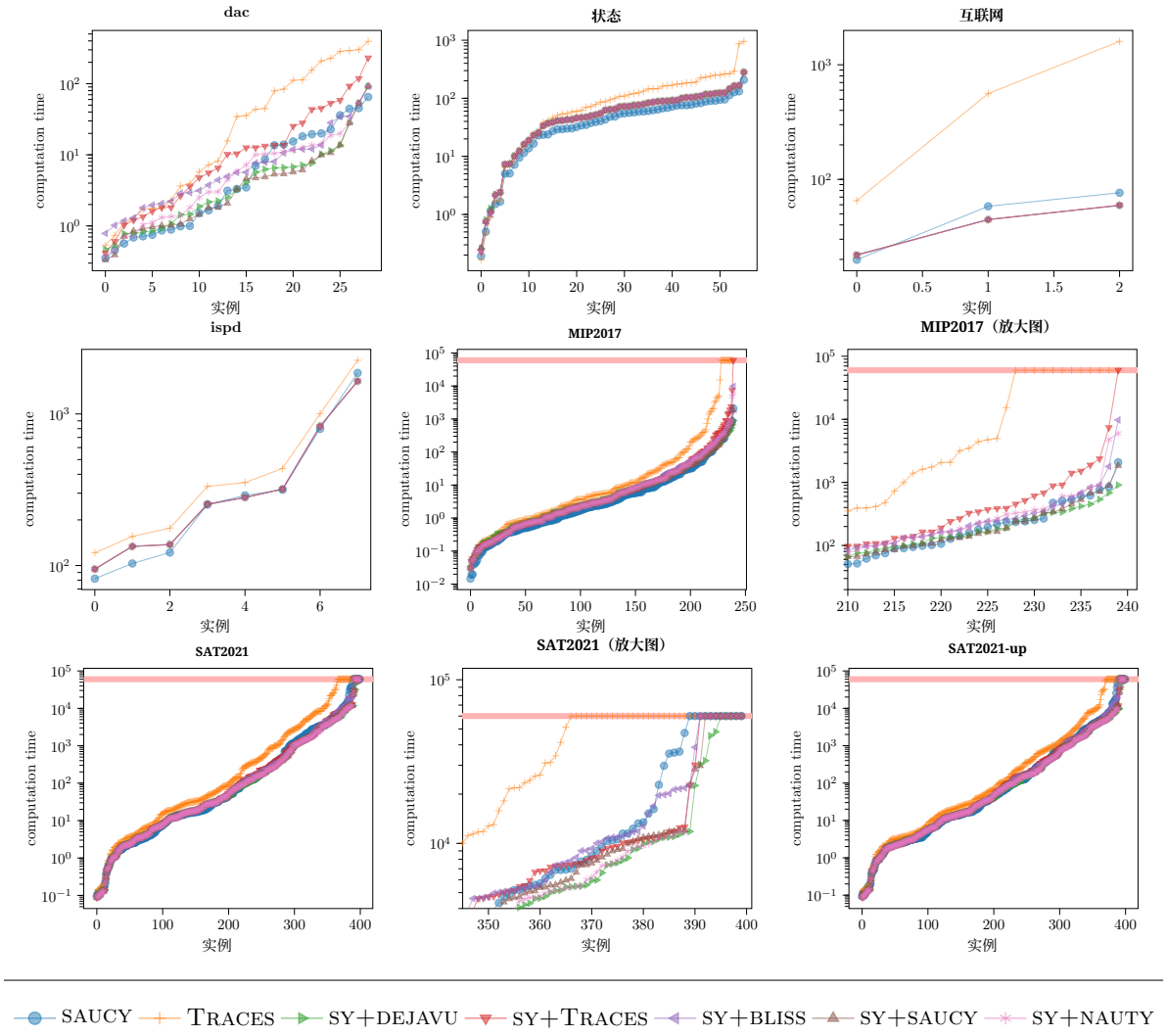


图 12：图 11 中各个集合的详细图表。红色条形表示超时。实例按运行时间排序。

在不使用 TRACES预处理器的情况下。使用的超时时间为 $60s$ （求解器内存耗尽时也如此）。

基准测试包含 SAUCY 发行版中的所有集合。我们还测试了 3 组当代实际图：MIP2017 集合包含源自混合整数规划库的图（见 [1]）。SAT2021 库包含源自 SAT 竞赛中 SAT 实例的图（见 2021 [4]）。SAT2021-up 集合与之类似。不过，这些实例首先经过了单元规则和纯文字规则的预处理（关于其重要性的讨论，见 [5]）。值得一提的是，SAT 集合包含许多大型图，其顶点数达数千万——尤其是其中最大的图，在所有受测集合中规模最大。

结果汇总于图 9 和图 10，以及图 11 和图 12。首先，我们观察到，先前的最先进水平（SAUCY）在除一组之外的所有集合上，都被若干使用预处理器的求解器超越。这表明，将我们的通用预处理器与不同求解器结合使用，可以达到最先进水平之上。此外，在大多数集合上，SAUCY 和 TRACES 应用预处理器后也明显加快了速度。

我们想指出，在 SAT2021 数据集中一些规模特别大的图上，DEJAVU 和 TRACES 确实会耗尽内存。因此，根据可用 RAM 的大小以及评估中对内存消耗的权重，其他求解器可能更合适。在所有 DEJAVU 耗尽内存的情况下，其他求解器也都会超时。无论如何，在所有与此相关的数据集中，SY+NAUTY 和SY+SAUCY 的表现也都优于 SAUCY。

进一步来说，我们经常观察到，SAUCY 在能够快速求解的实例上表现最佳，但在规模更大或求解速度更慢的实例上则表现不佳。我们认为，这可能表明预处理器实现本身仍有尚未发掘的底层优化潜力。

10 结论与未来发展

我们介绍了用于对称性检测的新型 SASSY 预处理器。我们证明，SASSY确实能加速最先进的求解器处理大型实用图。

未来可以为预处理器增加更多用于度为 2 的顶点消除的启发式方法，强化不变量，甚至实现更高效的实现方式，并调整现有启发式方法。鉴于我们观察到最先进的求解器对单元选择器的选择高度敏感，深入研究这一主题将很有意义。

致谢

我们感谢 Marc E. Pfetsch 和 Christopher Hojny 分享他们对对称性检测软件用户端的更多见解，并向我们提供 MIP2017 图。

参考文献

[1] MIPLIB 2017——混合整数规划库。https://https://miplib.zib. de/。

[2] nauty 和 Traces。http://pallini.di.uniroma1.it。

[3] sassy。https://github.com/markusa4/sassy。

[4] SAT竞赛2021。https://satcompetition.github.io/2021/。

[5] Markus Anders。《SAT 预处理器与对称性》。载于 Kuldeep S. Meel 和 Ofer Strichman 编，第*25*届国际可满足性测试理论与应用会议，*SAT 2022*，*2022*年*8*月*2*日至*5*日，以色列海法，第236卷，*LIPIcs*，第1:1–1:20页。Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，2022年。[6] Markus Anders 和 Pascal Schweitzer。《构建快速概率同构测试》。载于 Martin Farach-Colton 和 Sabine Storandt 编，算法工程与实验研讨会论文集，*ALENEX 2021*，线上会议，*2021*年*1*月*10*日至*11*日，第73–84页。SIAM，2021年。

[7] Markus Anders 和 Pascal Schweitzer。组合对称性的并行计算。载于 Petra Mutzel、Rasmus Pagh 和 Grzegorz Herman（编），第*29*届欧洲算法年会，*ESA 2021*，*2021*年*9*月*6*日至*8*日，葡萄牙里斯本（线上会议），LIPIcs 第204卷，第6:1–6:18页。Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，2021年。

[8] Markus Anders、Pascal Schweitzer 和 Florian Wetzels。最坏情况下之外的颜色细化算法比较性设计选择分析。载于 Nikhil Bansal、Emanuela Merelli 和 James Worrell（编），第*48*届自动机、语言与编程国际学术研讨会，*ICALP 2021*，*2021*年*7*月*12*日至*16*日，苏格兰格拉斯哥（线上会议），LIPIcs 第198卷，第15:1–15:15页。Schloss Dagstuhl - Leibniz-Zentrum f¨ur Informatik，2021年。

[9] Christoph Berkholz、Paul S. Bonsma 和 Martin Grohe。规范颜色细化复杂度的紧致上下界。*Theory Comput. Syst.*，60(4)：581–614，2017年。

[10] Armin Biere、Florian Lonsing 和 Martina Seidl。用于 QBF 的阻塞子句消除。载于 Nikolaj S. Bjørner 和 Viorica Sofronie-Stokkermans（编），自动演绎——*CADE-23*——第*23*届国际自动演绎会议，波兰弗罗茨瓦夫，*2011*年*7*月*31*日—*8*月*5*日。论文集，计算机科学讲义第6803卷，第101–115页。施普林格，2011年。

[11] Paul T. Darga、Mark H. Liffiton、Karem A. Sakallah 和 Igor L. Markov。利用结构进行 CNF 对称性检测。载于 Sharad Malik、Limor Fix 和 Andrew B. Kahng（编），第*41*届设计自动化会议论文集，*DAC 2004*，圣迭戈，加利福尼亚州，美国，*2004*年*6*月*7*日至*11*日，第530–534页。ACM，2004年。

[12] David D´eharbe、Pascal Fontaine、Stephan Merz 和 Bruno Woltzenlogel Paleo。《利用 SMT问题中的对称性》。载于 Nikolaj S. Bjørner 和 Viorica Sofronie-Stokkermans 编，《自动演绎——CADE-23——第23届国际自动演绎会议，波兰弗罗茨瓦夫，2011年7月31日—8月5日。论文集》，计算机科学讲义第6803卷，第222–236页。施普林格，2011年。

[13] Jo Devriendt、Bart Bogaerts 和 Maurice Bruynooghe。《对称解释学习：面向 SAT 的有效动态对称性处理》。载于 Serge Gaspers 和 Toby Walsh 编，《可满足性测试理论与应用——SAT 2017——第20届国际会议，澳大利亚维多利亚州墨尔本，2017年8月28日—9月1日。论文集》，计算机科学讲义第10491卷，第83–100页。施普林格，2017年。

[14] Niklas E´en 和 Armin Biere。通过变量和子句消除实现 SAT 的有效预处理。载于 Fahiem Bacchus 和 Toby Walsh （编），可满足性测试理论与应用，第八届国际会议，*SAT 2005*，英国圣安德鲁斯，*2005* 年*6*月*19*日—*23*日，论文集，计算机科学讲义第3569卷，第61–75页。施普林格，2005年。

[15] Ian P. Gent、Karen E. Petrie 和 Jean-Fran¸cois Puget。约束编程中的对称性。载于 Francesca Rossi、Peter van Beek 和 Toby Walsh（编），约束编程手册，人工智能基础第2卷，第329–376页。爱思唯尔，2006年。[16] Mark K. Goldberg。用于测试两个图同构性的非阶乘算法。离散应用数学，6(3):229–236，1983年。[17] Christopher Hojny 和 Marc E. Pfetsch。通过对称性破缺多面体进行对称性处理。载于 Ekrem Duman 和 Ali Fuat Alkaya（编），第*13*届科隆-特温特图与组合优化研讨会，土耳其伊斯坦布尔，*2015*年*5*月*26*日至*28*日，第63–66页，2015年。[18] Derek F. Holt、Bettina Eick 和 Eamonn A. O'Brien。计算群论手册。《离散数学及其应用》。Chapman and Hall/CRC，2005年。[19] Tommi A. Junttila 和 Petteri Kaski。用于规范标记的冲突传播和分量递归。载于 Alberto Marchetti-Spaccamela 和 Michael Segal（编），（计算机）系统中的算法理论与实践——第一届国际*ICST*会议，*TAPAS 2011*，意大利罗马，*2011*年*4*月*18*日至*20*日。论文集，计算机科学讲义第6595卷，第151–162页。施普林格，2011年。[20] Hadi Katebi、Karem A. Sakallah 和 Igor L. Markov。对称性与可满足性：更新。载于 Ofer Strichman 和 Stefan Szeider（编），可满足性测试理论与应用——*SAT 2010*——第*13*届国际会议，*SAT 2010*，英国爱丁堡，*2010*年*7*月*11*日至*14*日。论文集，计算机科学讲义第6175卷，第113–127页。施普林格，2010年。[21] Manuel Kauers 和 Martina Seidl。量化布尔公式的对称性。载于 Olaf Beyers- dorff 和 Christoph M. Wintersteiger（编），可满足性测试理论与应用——*SAT 2018*——第*21*届国际会议，*SAT 2018*，作为联合逻辑会议 *FloC 2018* 的一部分举行，英国牛津，*2018*年*7*月*9*日至*12*日，论文集，计算机科学讲义第10929卷，第199–216页。施普林格，2018年。[22] Sandra Kiefer、Ilia Ponomarenko 和 Pascal Schweitzer。平面图的 Weisfeiler-Leman 维数至多为3。*ACM*期刊，66(6):44:1–44:31，2019年。[23] Sandra Kiefer、Pascal Schweitzer 和 Erkal Selman。由带计数的逻辑识别的图。*ACM*计算逻辑汇刊，23(1):1:1–1:31，2022年。[24] Tuukka Korhonen、Jeremias Berg、Paul Saikko 和 Matti J¨arvisalo。Maxpre：扩展的最大可满足性预处理器。载于 Serge Gaspers 和 Toby Walsh（编），可满足性测试理论与应用——*SAT 2017*——第*20*届国际会议，澳大利亚维多利亚州墨尔本，*2017*年*8*月*28*日—*9*月*1*日，论文集，计算机科学讲义第10491卷，第449–456页。施普林格，2017年。[25] Fran¸cois Margot。整数线性规划中的对称性。载于 Michael J¨unger、Thomas M. Liebling、Denis Naddef、George L. Nemhauser、William R. Pulleyblank、Gerhard Reinelt、Giovanni Rinaldi 和 Laurence A. Wolsey（编），整数规划*50*年（*1958—2008*）——从早期发展到最先进水平，第647–686页。施普林格，2010年。[26] Brendan D. McKay 和 Adolfo Piperno。实用图同构，第二部分。符号计算杂志，60:94–112，2014年。

[27] Marc E. Pfetsch 和 Thomas Rehn。《混合整数规划对称性处理方法的计算比较》。数学规划计算，11(1):37–93，2019年。[28] Adolfo Piperno。《带加权边有向图的同构测试》。载于 Gianlorenzo D'Angelo（编），第*17*届国际实验算法研讨会，*SEA 2018*，*2018*年*6*月*27*日至*29*日，意大利拉奎拉，第103卷，*LIPIcs*，第30:1–30:13页。Schloss Dagstuhl - Leibniz 信息学中心，2018年。[29] Pascal Schweitzer 和 Daniel Wiebking。《一种统一的组合对象规范化算法设计方法》。载于 Moses Charikar 和 Edith Cohen（编），第*51*届*ACM SIGACT*年度计算理论研讨会论文集，*STOC 2019*，美国亚利桑那州菲尼克斯，*2019*年*6*月*23*日至*26*日，第1247–1258页。ACM，2019年。[30] ´Akos Seress。置换群算法。剑桥数学专著。剑桥大学出版社，2003年。

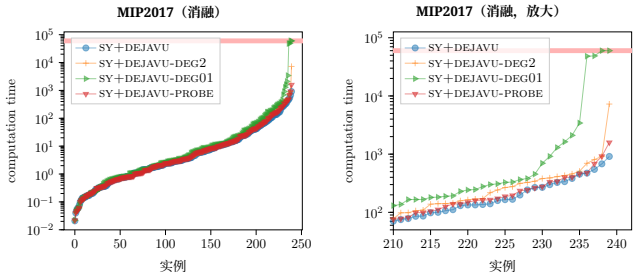


图 13： SY+DEJAVU 在 MIP2017 图上的消融研究（时间：9.15s、17.87s、234.47s、10.65s），超时为 60s。

附录

A 组合图中的离群项

在比较 TRACES预处理与未预处理实例的评估中，有一个特别突出的离群值。这个实例是一个有408个顶点的缩小版多足图。未进行预处理时，求解耗时为 $0.75s$；经过预处理后，求解耗时为 $6.3s$。乍看之下，这令人困惑：首先，预处理器在不到0.5毫秒内就完成了。预处理器也没有改变图——没有移除任何顶点或边——只是为其赋予最粗等价着色。不过，这种着色与 TRACES 为该图计算出的着色几乎相同。

唯一的区别是， TRACES 内部可能会以不同的方式命名颜色，例如，颜色 3 可能会被命名为颜色 6。虽然这并不会从结构上使图变得更难或更容易，但启发式方法在内部可能总是会选择“编号最小的最大颜色”（这类似于 SAT 求解器中的变量排序）。因此，重命名颜色可能会影响求解器所做的决策。不过，使用“编号最小”的颜色通常并非有意为之。事实上，如果我们简单地反转颜色顺序，这个图确实能在 $0.12s$内求解。我们还想指出， TRACES 在该图上的 $0.75s$ 已经比所有其他求解器都慢。

在 [7], 中也有人指出，单元选择器的选择会显著影响收缩多足图的集合。我们认为，解决这一问题的办法是做出结构上更好的选择，而这确实与预处理器的作用关系不大。

我们还想提及，在其他测试中，我们也发现，对于大多数求解器，图经过预处理后也会以同样的方式变得更容易求解。

B 消融研究

在图 13 中，我们评估了 DEJAVU 在 MIP2017 数据集上分别应用各项预处理技术的效果。具体做法是运行配置 SY+DEJAVU，但针对每项技术分别运行一次，并停用相应技术。例如，SY+DEJAVU-DEG2 运行时 SY+DEJAVU 不使用任何度为 2 的顶点删除技术。数据显示，每项技术都能有效缩短运行时间。影响最大的是消除度为 0 和 1 的顶点，其次是消除度为 2 的顶点。即使不使用探测，求解该数据集也比以往最先进水平更快（参见图 11）；不过，在其他技术的基础上使用探测仍有帮助。