arXiv:2602.05503v1 [cs.DB] 5 Feb 2026

# PG 约束下的属性图修复

Christopher Spinrath 里昂第一大学、里昂法国国家科学研究中心Liris实验室，法国
christopher.spinrath@liris.cnrs.fr

Angela Bonifati 里昂第一大学、里昂法国国家科学研究中心Liris实验室与IUF，里昂，法国
angela.bonifati@univ-lyon1.fr

Rachid Echahed 法国国家科学研究中心LIG实验室、格勒诺布尔阿尔卑斯大学 格勒诺布尔，法国
rachid.echahed@imag.fr

## 摘要

近期针对图数据库的标准化工作催生了 GQL 和 SQL/PGQ 等标准查询语言，以及属性图约束（PG约束）等约束语言。本文着手研究 PG 约束下的属性图修复。我们识别出 PG约束的一个重要子集，该子集编码了否定约束并将递归作为关键特征，同时仍支持基于自动机的错误结构分析。我们提出了一个针对这些约束的完整修复流程，用于修复属性图；该流程涉及图拓扑的变更，并导致节点、边以及可选的标签删除。我们研究了三种修复过程的算法策略，分别基于整数线性规划（ILP）、朴素算法和线性规划引导的贪心算法。在多个真实世界数据集上的实验表明，与节点/边删除相比，采用标签删除可使删除量减少 59%。此外，线性规划引导的贪心算法在保持相同质量的同时，相较于整数线性规划策略可实现最高达 97% 的运行时间优势。

**相关版本：**
本文（不含附录）已被 PVLDB 第19卷录用，并将在第52届超大型数据库国际会议（VLDB 2026）上发表。

**工件可用性：**
源代码、数据和/或其他工件已在 https://doi.org/10.5281/zenodo.18301604上提供。

## 1 引言

处理大型数据集时，修复错误和不一致是项基本任务。在数据集成场景中，或数据库的要求（例如模式）发生变化时，自然会出现此类应用。因此，针对各种数据库模型的错误修复已在文献中得到广泛研究 [13]。

涵盖查询语言（即 GQL 和 SQL/PGQ） [18, 30], 以及模式和约束语言（例如 PG-Schema 和 PG-Keys） [3, 4], 的属性图语言最近已实现标准化。

据我们所知，针对这些模式和约束语言的属性图修复技术尚未得到研究。我们着手研究定性错误修复 [13]，针对的是属性图，也就是说，是相对于约束进行错误修复；更准确地说，是针对 PG-Schema 形式体系中的一部分 PG约束 [3, 4], ，它们扩展了 PG-Keys。

版权所有 ©2026年 Christopher Spinrath、Angela Bonifati 和 Rachid Echahed 本作品采用知识共享“署名-非商业性使[illegible] 4.0 国际”许可。



为了说明我们的研究动机，我们通过一个贯穿全文的示例来展示属性图和 PG 约束。

示例 *1.1.* 图 1 展示了一个属性图 $G_e$ 该图对一个组织及其人员、任务和文档进行建模，用于规划活动。人员可以执行任务，而任务会引用其所需的文档。节点和边都可以有多个标签，也可以配有键值对，称为属性。例如，节点 $d_3$ 有两个标签：文档和重要，以及两个属性：#页数和访问_级别。

请看以下针对属性图 $G_e$的约束。

> “如果某个人从事一项已开始的任务，而该任务直接或间接地（即递归地）引用了一份重要文档，那么此人的访问级别至少应与被引用文档所要求的访问级别一样高。”

该约束在其前置条件中描述了一个图模式：其中，“某个人从事一项任务”指的是一个带有 person 标签的节点，以及一个带有 task 标签、并通过一条标记为 works_on 的边与之相连的节点。该部分模式在 $G_e$ 中的匹配由节点 $p_1$、 $t_1$ 和边 $w_1$组成。所描述的模式还扩展到该任务直接或间接引用的文档。上述部分模式的匹配可以通过多种方式扩展： $d_1$、 $d_2$ 和 $d_3$ 都是被 $t_1$ 引用的文档，其中 $d_2$ 和 $d_3$ 是被间接引用的。一般来说，引用链的长度可能任意。然而，文档 $d_2$ 没有标记为 important，因此不应匹配。

最后，该约束比较属性值，即开始属性 $t_1$ ，以断言任务已经开始，以及访问_级别属性 $p_1$ ，并根据匹配到的节点是 $d_1$ 还是 $d_3$。图*PG* 约束 [4] 对该约束的一种可能规范如图1所示。

$G_e$违反了上述约束，因为 $p_1$、 $w_1$、 $t_1$、 $r_1$、 $d_1$、 $r_4$ 和 $d_3$（其中 $d_3$ 是相关的重要文档）构成了一个匹配，任务 $t_1$ 已经开始，但 $d_3$ 的访问级别高于 $p_1$。

贡献。我们将正则图模式演算（*RGPC*）约束确定为 PG约束的一个子集，并为其模式引入一种自动机模型，以推理错误的结构。同时，我们的子集涵盖了 PG约束的主要特性；尤其是递归、对节点和边标签的复杂限制，以及节点属性的可比较性。它们可以表达否定约束，从而捕捉包括函数依赖在内的多种约束，其中包括示例1.1中的约束。为指定模式，我们使用图模式演算 GPC 的一个子集 [26], ，该子集涵盖了 GQL 和 SQL/PG 中模式的核心特性。

在数据库修复领域，文献中研究了多种修复模型 [20], ，包括添加新数据 [5], 、更改数据值 [8, 34, 43] （即属性的值）以及删除 [12]。修复模型的选择取决于具体场景。

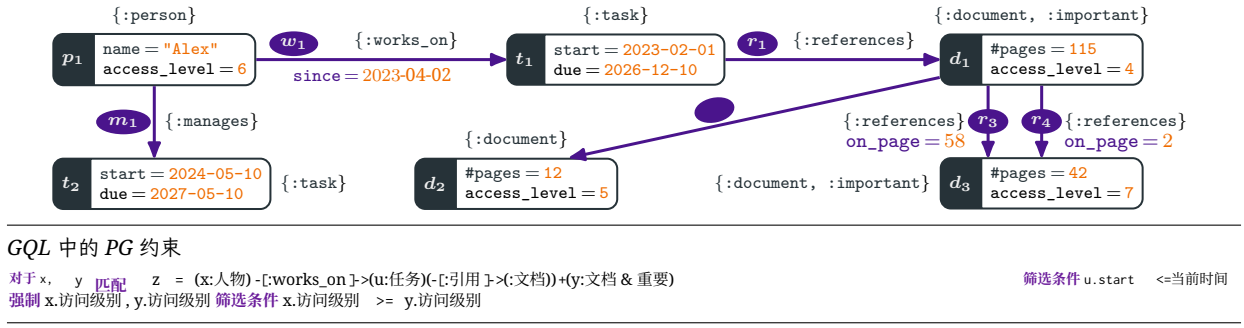



**图 1：一个对包含人员、任务和文档的组织进行建模的属性图，以及一个 PG 约束**

本文假设采用删除修复模型 [例如参见12, 20, 33]，在这种模型中，通过删除图中的对象来修复属性图，即节点、边或标签。在某些应用场景中，删除作为修复模型是合理的，因为插入新数据并不可取，或风险过高，例如出于隐私或安全方面的考虑。更具体地说，在社交网络中，添加友谊关系可能会暴露隐私数据，而删除友谊关系则可以纠正这一问题。在对供应链进行建模的属性图中，插入制造商与产品之间的供应_来源于关系，而该制造商实际上无法供应该产品，可能会给公司带来灾难性后果；删除这种关系则能凸显产品无法供应的问题。在图 1 所示属性图所建模的场景中，提高（即更改）人物或文档的访问级别会带来安全风险。请回想，图 1 所示的属性图不满足示例 1.1 中的约束，因为由 $p_1$ 表示的人物在任务 $t_1$ 上工作，而该任务引用了重要文档 $d_3$。但是，$p_1$ 的访问级别不足以访问 $d_3$。修复该属性图的一种方法是删除边 $w_1$，从而移除 $p_1$ 在 $t_1$上工作的关系。另一种方法是从重要中移除标签 $d_3$。

本文提出一条完整的流水线（参见图 2），其中包含不同的图修复选项。该修复方法采用整体方法，即同时检测并修复所有错误。这确保尽可能少地移除对象，并尊重它们之间的关系：例如，删除一个节点意味着删除其所有关联边。然而，处理多个错误——尤其是涉及较长的任意长度路径而非固定长度元组的错误——可能非常耗时。为缓解这一问题，我们提出三种算法，用于解决确定待删除对象这一底层组合问题。第一种是朴素贪心算法，它依次修复每个错误，即从中删除一个对象，除非该错误已通过先前的删除得到解决。第二种将问题表述为整数线性规划（整数线性规划），并使用求解器得出待删除对象的最优集合。第三种是线性规划引导的贪心算法，它利用整数线性规划的 LP 松弛的非整数解来引导贪心算法。

最后，我们通过在真实世界数据集上开展实验，评估修复流程的有效性，并分析不同算法和选项之间的权衡。实验结果表明，线性规划引导的贪心算法能够优于其他算法，同时保持与整数线性规划策略相当的质量，即删除数量。允许标签删除最多可减少 59% 的删除数量，但会增加运行时间（最多增加 5×）。此外，实验结果还表明，我们的算法的近似版本最多可提速 89%，同时仍能给出良好的近似结果，甚至实现修复。

结构。在第 2节中，我们回顾属性图的基础知识，并介绍图模式演算（GPC）的一个子集。随后，我们在第 3节中引入 RGPC 约束。在第 4 节中，我们介绍用于错误修复的流水线和算法，并在第 5 节中介绍实验。最后，我们在第 6节讨论其他相关工作，并在第 7节总结。

## 2 预备知识

本节介绍属性图中使用的数据模型的基础知识，并介绍模式演算 GPC 的一个子集 [26]，我们将在第 3节中将其用作约束的组成部分。

### 2.1 属性图

用 L、K、V 和 I 表示可数无限且两两不相交的集合，分别表示标签、键（属性的）、属性值和标识符（简称*id*）。

一个属性图是一个元组 $G = (N, E, \rho, \lambda, \pi)$，其中

- $N \subset \mathrm{I}$ 是一个有限的节点标识符集合；
- $E \subset \mathrm{I}$ 是一个与 $N$ 不相交的有限边标识符集合；
- $\rho$: $E \to (N \times N)$ 定义所有边的端点，即源和目标；
- $\lambda$: $(N \cup E) \to 2^{\mathcal{L}}$ 是一个标记函数，它为节点和边标识符分配有限个标签；并且
- $\pi$: $((N \cup E) \times \mathrm{K}) \to \mathrm{V}$ 是一个偏函数，为（节点或边）标识符与键构成的对分配值。

为方便起见，我们将节点和边标识符统称为属性图中的节点和边。此外，我们用对象一词来指代节点、边，以及特定节点或边上的标签。

例如，图属性图中的节点 $G_e$ 如图 1 所示，分别为 $p_1$、$t_1$、$t_2$、$d_1$、$d_2$、$d_3$；边为 $w_1$、$m_1$、$r_1$、$r_2$、$r_3$、$r_4$。边 $w_1$ 的端点为 $p_1$（源）和 $t_1$（目标）。需要注意的是，节点或边可能没有任何标签。在这种情况下，$\lambda$ 会分配空集（标签集合）。

属性图 $G = (N, E, \rho, \lambda, \pi)$ 中的一条 路径 是一个非空交替序列 $v_0 e_1 v_1 \dots e_n v_n$，其中 $v_0, \dots, v_n \in N$是节点，$e_1, \dots, e_n \in E$是边，使得 $\rho(e_i) = (v_{i-1}, v_i)$ 对所有成立

$i \in \{1, \ldots, n\}$。例如，序列 $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ 是 $G_e$ 中的一条路径。我们强调，每条路径都以节点开始并以节点结束。路径 $v_0 e_1 v_1 \ldots e_n v_n$ 的长度 是边的数量 $n$。空路径恰好由一个节点组成，长度为 0。若 $v_n = v'_0$ 成立，则两条路径 $v_0 e_1 \ldots e_n v_n$ 与 $v'_0 e'_1 \ldots e'_m v'_m$ 可以拼接 。所得路径即为 $v_0 e_1 \ldots e_n v_n e'_1 \ldots e'_m v'_m$.。路径 $P = v_0 e_1 v_1 \ldots e_n v_n$ 的轨迹 $\lambda(P)$ 为 $\lambda(P) = \lambda(v_0)\lambda(e_1)\lambda(v_1) \ldots \lambda(e_n)\lambda(v_n)$。例如，路径 $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ 的轨迹为

{人物}{工作_于}{任务}{引用}{文档，重要}{引用}{文档，重要}。

## 2.2 正则图模式演算

下面我们将介绍 GPC 的一个真子集——一种图模式演算的语法和语义。它涵盖了 GQL 和 SQL/PGQ [26] 共同采用的模式子语言的核心特征。我们通过示例介绍这个子集，并将其称为 *RGPC*（即 正则 *GPC*）。

示例 *2.1*。 以下 RGPC 模式匹配所有满足以下条件的节点对 $x, y$：$x$ 带有文档标签，并且存在一条从 $x$ 到 $y$ 的路径 $z$，其上的每条边都带有引用标签。

$$z = (x : \mathsf{document})\ [\xrightarrow{:\mathsf{references}}]^*\ (y)$$

我们将自底向上地说明示例 2.1 中模式的组成部分以及一些变体。RGPC 模式的基本构件是节点模式和边模式。节点模式的形式为（$x : \varphi$），其中 $x$ 是节点变量，$\varphi$ 是用于限制匹配该模式的节点标签的标签表达式。$x$ 和 $\varphi$ 都是可选的。示例 2.1 中的节点模式是（$x$: document）和（$y$）。在后一个模式中，$y$ 的标签不受限制，该节点可以带有任意标签。例如，不含变量的节点模式为（: a）。如果同时省略 $x$ 和 $\varphi$，则只写作 ()。一般来说，$\varphi$ 可以是一个命题公式，用运算符 ∧（合取）、∨（析取）和 ¬（否定）连接集合 L 中的标签。例如，（: (a ∧ b) ∨ ¬c）匹配所有带有标签 a 和 b，或者不带有标签 c 的节点。

与节点模式类似，边模式的形式为: $\xrightarrow{\varphi}$，不过我们不考虑边模式中的变量。在示例2.1中，有一个边模式，即:引用 $\longrightarrow$。

路径模式。节点模式和边模式可以通过连接、并集和重复组合成路径模式。

例如，两个节点模式和一个边模式的连接如下：

$$(x : \mathsf{document}) \xrightarrow{:\mathsf{references}} (y)$$

它匹配所有满足以下条件的节点对 $x, y$：$x$ 的标签为文档，并且存在一条从 $x$ 到 $y$、标签为引用的边。

如果两个模式不包含节点变量，就可以构造它们的并集。例如，:a $\longrightarrow \cup$ :c$\longrightarrow$ :c$\longrightarrow$ 表示存在一条标签为 a 的边，或者存在一条长度为二、所有边的标签均为 c 的路径。

对于重复，我们考虑经典的克林星运算符。例如，$[$:a $\longrightarrow]^*$ 表示由: a $\longrightarrow$ 所指定的路径可以任意多次重复（包括 0 次）。我们也将 $[$:a$\longrightarrow]^+$作为: a $\longrightarrow [$:a$\longrightarrow]^*$的简写。与并集类似，重复仅限于不包含任何节点变量的模式。此外，重复下方的模式不得匹配长度为 0 的路径。

当然，连接、并集和重复可以嵌套。为提高可读性（圆括号用于节点模式），我们使用方括号来消除歧义。

图模式。最后，图模式是路径模式的合取，并与两两不同的路径变量相关联。[1] 例如，以下图模式由两个路径模式组成

$$z_a = (x : \mathsf{b}) \xrightarrow{:\mathsf{a}} (y),\ z_c = (y) \xrightarrow{:\mathsf{c}}{}^{+}\ (x)$$

包含变量 $z_a$ 和 $z_c$。它指定属性图中存在节点 $x$，$y$，使得 $x$ 的标签为 b；存在一条标签为 $a$ 的边（$x$，$y$）；并且存在一条从 $y$ 到 $x$ 的路径，其上的每条边的标签都是 $c$。需要指出的是，路径变量与节点变量互不相交。

示例 *2.2*。 考虑以下图模式，其中 p、w、t、d、r 和 i 分别是标签 person、works_on、task、document、references 和 important 的缩写。

$$z = (x : \mathsf{p}) \xrightarrow{:\mathsf{w}} (:\mathsf{t})\Big[\xrightarrow{:\mathsf{r}} (:\mathsf{d})\Big]^+ (y : \mathsf{d} \wedge \mathsf{i})$$

它指定了示例 1 1中的拓扑：person $x$ 处理一项任务，而该任务（可能通过间接路径）引用了一份重要文档 $y$。注意，从该任务到 $y$ 的子路径上的所有节点都必须标记为document，但表示该任务的第一个节点除外。

语义。 属性图$G$ 中（图）模式 $q$ 的匹配 是一个映射 $\mu$，它分别将 $q$ 的节点变量和路径变量映射到 $G$中的节点和路径，并由此构成 $G$上 $q$ 的一个答案。关于形式化定义，请参阅文献 [26]。

示例 *2.3*。 示例 2.2 中的模式在图 1所示属性图中的一个匹配是映射 $\mu$，其中 $\mu(x) = p_1$，$\mu\ y$ $\mu\ z$ $() = d_3$，以及 $() = p_1 w_1 t_1 r_1 d_1 r_3 d_3$。

# 3 属性图约束

预备知识已经就绪，我们现在可以介绍本文所研究的约束。我们首先通过一个示例来说明这一思想，然后再使相关术语更加精确。

示例 *3.1*. 我们将以下约束作为本节的贯穿示例：“每份文档的访问级别都高于它所引用（直接或间接）的、自2020年以来未被编辑过的任何其他文档。”

约束的核心成分是 RGPC 图模式，它融入了既有图查询语言（如 GQL 和 SQL/PGQ）的特性，尤其是递归。这使我们能够表达示例 3.1中所描述的拓扑：该约束作用于所有文档对，记为 $x$与 $y$，使得 $x$（直接或间接）引用 $y$。以下 RGPC 模式对该拓扑进行建模。

$$z = (x : \mathsf{document})\ [\xrightarrow{:\mathsf{references}}]^+\ (y : \mathsf{document})$$

RGPC 模式再辅以另外两个要素，即可得到一条约束。这两个要素都可以引用节点变量。

第一个可以理解为对模式匹配结果施加的额外筛选条件。例如，谓词 $x \neq y$可确保 $x$和 $y$在上述模式中匹配到不同的文档。此外，示例 3.1 中的约束只影响 $y$ 自 2020年以来未被编辑过的文档document

[1]在 GPC 的原始定义中，路径变量是可选的，但在这里它们是必需的。它们也不必互不相同。我们要求它们互不相同，以避免“隐藏的”路径相等约束。

具有记录上次编辑时间的属性，即时间戳，则可以通过谓词 $y$.edited $\leq$ 2020-12-31来确保这一点。通常，可以提供任意数量的谓词（也可以不提供），此时所有谓词都必须成立，匹配结果才能通过筛选条件。

最后一个组成部分同样由谓词和状态构成，这些条件必须对每一个通过筛选条件的模式匹配成立。在我们的示例中，该条件是由谓词 $x$.access_level $\geq$ $y$.access_level 构成的集合。

总体而言，示例 3.1 中描述的约束可使用示例 2.2中的简写表示如下。

$$z = (x:\mathsf{d})\ [\xrightarrow{:\mathsf{r}}]^{+}\ (y:\mathsf{d}); \{x \neq y,\ y.\mathtt{edited} \leq \mathtt{2020\text{-}12\text{-}31}\} \Rightarrow \{x.\mathtt{access_level} \geq y.\mathtt{access_level}\}$$

一般而言，本文所研究的约束具有形状$q$；$F \Rightarrow C$其中 $q$ 是一个 RGPC 图模式，而 $F$和 $C$ 分别是对约束的筛选条件与条件进行建模的谓词集合。更确切地说，我们考虑 GQL [30, 第 19.3] 节中的所有比较谓词：$x_i = y_j$、$x_i \neq y_j$、$x_i$.propA $\oplus$ $y_j$.propB，以及 $x_i$.propA $\oplus$ $c$，其中 $x_i$和$y_j$是出现在 $q$中的节点变量，$\oplus \in \{=, \neq, \leq, \geq, <, >\}$，并且 $c$是来自 V 的一个值。

这些谓词的语义与通常情况相同，值得注意的是，所有提及的属性都必须存在。例如，上述约束中的筛选条件和条件分别意味着 edited and access_level 属性的存在。形式化定义见附录 A.2。如果一个匹配 $\mu$ 满足集合中的所有谓词，则称其满足一组谓词。属性图 $G$满足约束 $q$；$F \Rightarrow C$当且仅当 $\mu$中 $G$ 的每个匹配 $q$ 满足$C$或不满足 $F$。

示例 *3.2*。 以下约束 $\sigma$ 在示例 1.1中作了非正式描述。

$$\underbrace{z = (x:\mathsf{p}) \xrightarrow{:\mathsf{w}} (u:\mathsf{t})\Big[\xrightarrow{:\mathsf{r}} (:\mathsf{d})\Big]^{+}(y:\mathsf{d} \wedge \mathsf{i});}_{q}$$

$$\{u.\text{开始} \leq \text{现在}\} \Rightarrow \{x.\text{访问_级别} \geq y.\text{访问_级别}\}$$

这里 $q$ 是示例 2.2中说明的 RGPC 模式，只是额外增加了节点变量 $u$。我们用“now”作为当前时间戳的占位符。图 1 所示的属性图不满足该约束，因为示例 2.3 中的匹配满足筛选条件（任务的开始日期已过），但不满足 $\{x.\mathtt{access_level} \geq y.\mathtt{access_level}\}$。事实上，$x$与 $p_1$ $y$ $d_3$ $p_1$ _ $= 6 \not\geq$ 匹配，$7 = d_3$ 且与.....匹配，并且我们有 .accesslevel .accesslevel

如引言中所述，例3.2 中的约束 $\sigma$ 可表示为图1中的 PG 约束：RGPC 模式被转换为类似 ASCII 艺术图的 GQL 模式，而筛选条件和后件则用FILTER 子句表示。类似地，所有 RGPC 约束都可以写成 PG 约束。

示例 *3.3.* 到目前为止，我们的示例都是蕴含依赖，它们是否定约束的一个子类。利用不可满足谓词 $x \neq x$（我们简单地将其记作 false），还可以表示不属于这一子类的否定约束。例如，以下否定约束指出文档之间不存在循环引用。

$$(x:\mathsf{d})\Big[\xrightarrow{:\mathsf{r}} (:\mathsf{d})\Big]^{+}(x); \emptyset \Rightarrow \{\mathtt{false}\}$$

# 4 修复属性图

本节将介绍我们的修复流程，如图 2 所示。我们首先在第 ***4.1*** 节介绍 属性图的修复。在第 4.2 节，我们介绍基础流水线，也就是不包含任何可选步骤的流水线。我们将在第 4.3 节和 4.4 节介绍这些可选步骤。

## 4.1 属性图修复

如引言中所述，受某些应用场景的启发——在这些场景中，由于隐私或安全等原因，更改或添加对象的风险过高——我们通过删除属性图中的对象（即节点、边或标签）来修复属性图。

例 *4.1.* 考虑属性图 $G_{\mathrm{e}}$，如图 1 所示，以及约束 $\sigma$（见例 3.2）。属性图 $G_{\mathrm{e}}$ 不满足该约束，因为“Alex”在任务 $t_1$ 上工作，而该任务通过文档 $d_1$间接引用了重要文档 $d_3$。然而，“Alex”的访问级别严格低于 $d_3$。

要修复 $G_{\mathrm{e}}$，在删除修复模型下，只需删除节点 $p_1$、$t_1$、$d_1$ 或 $d_3$ 中的一个，或者删除它们之间的边 $w_1$ 或 $r_1$。注意，删除 $r_3$ 或 $r_4$ 都不足以修复（但同时删除两者则可以）。此外，这些操作都会改变 $G_{\mathrm{e}}$的拓扑。

另一种选择是删除节点或边的标签，例如，在此处，只需移除 $d_3$上的标签:重要即可。这种做法完全不会改变图的拓扑。

为了恰当地指定修复，我们将使用子图。直观地说，子图是通过从给定的属性图中删除节点、边、标签（或属性）而得到的属性图。

定义 4.2（子图）。设 $G = (N, E, \rho, \lambda, \pi)$ 是一个属性图。若属性图 $G' = (N', E', \rho', \lambda', \pi')$ 满足以下五个条件，则称其为 $G$ 的子图：(i) $N' \subseteq N$，(ii) $E' \subseteq E$，(iii) 对所有 $o' \in E'$，$\rho'(o') = \rho(o') \cap (N' \times N')$，(iv) 对所有 $o' \in N' \cup E'$，$\lambda'(o') \subseteq \lambda(o')$，以及 (v) 若属性键 $k$ 的 $\pi'(o', k)$ 有定义，且 $o' \in N' \cup E'$，则 $\pi(o', k)$ 和 $\pi'(o', k) = \pi(o', k)$ 也有定义。也就是说，$G'$ 中的所有节点、边、标签和属性（及其值）也都存在于 $G$ 中。若 $G'$ 中的所有节点和边都具有与 $G$ 中完全相同的标签和属性键值对，即对所有 $o' \in N' \cup E'$ 和属性键 $k$，$\lambda'(o') = \lambda(o')$ 且 $\pi'(o', k) = \pi(o', k)$，则称 $G'$ 为 $G$ 的拓扑子图。如果 $G \neq G'$ 成立，则 $G'$ 是 $G$ 的真子图。

例如，从节点 $d_3$ 中移除标签：重要，该节点位于属性图 $G_{\mathrm{e}}$ 中，如图1 所示，得到一个真子图，记作 $G_{\mathrm{e}}^{\star}$。子图 $G_{\mathrm{e}}^{\star}$并非拓扑子图，因为节点 $d_3$ 也存在于原图 $G_{\mathrm{e}}$ 中，但是在 $d_3$ 中的标签与 $G_{\mathrm{e}}^{\star}$中的标签不同 $G_{\mathrm{e}}$。另一个例子是从 $G_{\mathrm{e}}$ 删除边 $r_3$ 和 $r_4$ 后得到的拓扑子图。

要成为修复，子图必须满足所有约束，并且依照先前工作的思路 [参见例如12，定义 2。2],它应当是极大的，即不存在不必要的删除。

定义 *4.3*（修复）。给定一个属性图 $G$ 和一组 $\Sigma$ 约束，$G$的一个修复是 $G$的一个子图 $G'$，满足

a) 满足 $\Sigma$；并且

b) 是极大的，也就是说，不存在这样的子图 $G''$ 属于 $G$，使得$G'$ 是 $G''$ 的真子图，且 $G''$ 满足 $\Sigma$。

当子图 $G_a$ 属于 $G$ 并满足条件a)，但不一定满足条件b)时，我们称 $G_a$ 为一个近似修复。拓扑（近似）修复的定义与（近似）修复类似，不同之处在于涉及的所有子图都是拓扑子图。

例 *4.4*（例 4.1的延续）。令 $G'_e$ 为属性图 $G_e$ 删除边 $r_1$ 后得到的子图，该边位于 $t_1$ 与 $d_1$之间。我们注意到，$G'_e$ 是一个拓扑修复。但它不是修复，因为从 $G''_e$ 中删除 $G_e$ （仅）:references 标签后得到的子图 $r_1$ 同样是一个修复，而 $G'_e$ 是 $G''_e$ 的子图。因此，定义4.3的条件b）被违反。不过，它是一个近似修复，因为它满足条件a）。

同样，删除节点 $d_3$ 并不是修复，因为只需删除 :document 或 :important 标签即可。请注意，删除节点总是意味着同时删除涉及该节点的所有边：否则，结果就不是定义良好的属性图。因此，删除 $d_3$ 会导致删除边$r_3$ 和 $r_4$。但这样得到的子图也<style id='9'>不是拓扑修复：只要不把边加回来，就可以将 $d_3$ 加回去。我们将在后文更详细地讨论这一影响。

## 4.2 基础管道

现在，我们可以介绍修复流程了，该流程如图 2 所示。它以一组 Σ RGPC 约束和一个属性图 $G$ 作为输入，包含 6 个步骤，最后一步执行删除操作。其中两个步骤是可选的，即步骤 2 和 3：它们分别会导致标签删除，以及在质量和运行时间之间实现可控的权衡。接下来，我们首先讨论由步骤 基础流水线，即步骤 1、4、5和 6组成。

错误检索（步骤 *1*）。直观地说，我们将错误理解为一组对象，它们表明某个约束未被属性图满足。由于 RGPC 约束的所有路径模式都与一个路径变量相关联，因此可以从某个匹配的取值范围中获得错误：该匹配满足筛选条件，但不满足约束的后件。

再来看示例 3.2 中的约束以及图 1 所示的属性图。该属性图不满足此约束，因为该约束模式有两个匹配满足筛选条件，却不满足约束的条件：第一个匹配对应路径 $p_1w_1t_1r_1d_1r_3d_3$（参见示例 2.3），第二个匹配对应路径 $p_1w_1t_1r_1d_1r_4d_3$。如图 3 (1)、(2) 所示，这些路径上出现的节点和边集合，即 $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ 和 $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$，构成了拓扑错误。

定义 *4.5*（拓扑错误）。设 $G = (N, E, \rho, \lambda, \pi)$ 是一个属性图，$q$；$F \Rightarrow C$是一个包含路径变量 $z_1, \ldots\ldots, z_k$ 的约束，$\mu$是 $G$ 中 $q$ 的一个匹配，它表明该匹配不满足此约束。集合 $O_\mu$ 包含所有满足以下条件的节点和边：$o \in N \cup E$ 存在一个 $i$、$1 \le i \le k$，使得 $o$ 出现在 $\mu(z_i)$ 中。该集合称为 $G$ 中此约束的拓扑错误。

通过图查询进行错误检测的方法已在先前的工作中介绍过 [4]。我们将这种方法付诸实践，并通过将每个约束重写为一个错误查询，将错误检测和检索工作交由图数据库系统完成。该查询会找出所有涉及错误的路径。例如，示例3.2 中的约束被重写为以下 GQL 错误查询。

```
MATCH z  =(x:人物) -[: 工作_于 ]->(u:任务) (-[: 引用 ]->(:文档))+(y:文档 &
```

超图构造（步骤 *4*）。修复属性图时，思路是从每个错误中选取一个对象并将其删除。由于每个错误都对应一个匹配，删除其中一个对象就足以消除该匹配；因此，所得属性图满足该约束（组）。然而，结果不一定是修复。例如，分别从上文讨论的错误$O_1$ 和 $O_2$ 中选取 $r_1$ 和 $r_4$，并不能得到修复，因为可以重新添加 $r_4$ 而不违反约束。这是因为 $r_1$ 同时出现在两个错误中。我们已在示例4.4 中讨论过，删除节点也可能产生类似效果。

为了处理这一问题，可以将一组（拓扑）错误理解为一个超图，称为冲突超图。一个超图是一个二元组$\mathcal{H} = (\mathcal{W}, \mathcal{E})$，由顶点集合$\mathcal{W}$ 和超边集合[2] 组成，其中$\mathcal{E} \subseteq 2^{\mathcal{W}}$ 表示超边集合。我们的运行示例对应的冲突超图如图3 （3）所示。

定义 *4.6*（拓扑冲突超图）。给定一个属性图 $G = (N, E, \rho, \lambda, \pi)$、一组 Σ约束，$G$的拓扑冲突超图相对于 Σ是超图 $\mathrm{H} = (\mathrm{W}, \mathrm{E})$，其边恰好是 Σ 中所有约束的全部拓扑错误，并且 $\mathrm{W} = \bigcup_{O \in \mathcal{E}} O$ 是这些边的并集。

计算修复（步骤 *5c*）。要修复属性图，一种思路是从冲突超图中选择中间度较高的顶点，即参与大量错误的顶点。同时，我们希望选择数量最少的顶点。这相当于计算一个极小顶点覆盖：

超图 $\mathrm{H} = (\mathrm{W}, \mathrm{E})$ 的顶点覆盖 是一个集合 $V \subseteq \mathrm{W}$，使得 $V \cap O \neq \emptyset$，对每个 $O \in \mathrm{E}$ 都成立。如果一个顶点覆盖 $V$不存在更小的顶点覆盖 $V'$，满足 $V' \subsetneq V$，则称其为极小顶点覆盖。

直观地说，极小顶点覆盖告诉我们要删除哪些对象才能得到一个修复。例如，上文中包含超边 $O_1$ 和 $O_2$ 的冲突超图的顶点覆盖是 $V = \{r_1\}$。显然，$V$的任何子集都不是顶点覆盖，而删除 $r_1$ 即可得到一个修复，正如上文所述。集合 $V \cup \{r_4\} = \{r_1, r_4\}$ 不是极小顶点覆盖，因为它包含 $V$。实际上，删除 $r_1$和 $r_4$并不能得到修复，原因相同：可以将 $r_4$ 加回来。

然而，这并未解决节点删除会导致其关联边也被删除的问题。例如，$\{d_3\}$ 是一个极小顶点覆盖，但如例 4.4 所述，删除 $d_3$ 意味着删除 $r_3$ 和 $r_4$，因此无法得到修复。因此，除非确有必要（例如错误仅由节点构成），否则应避免删除节点。为此，我们为冲突超图 H 的每个顶点赋予一个权重。属性图中节点 $n$ 的权重 $w(n)$ 等于与 $n$ 关联的边数加一。边 $e$ 的权重 $w(e)$ 则为 1。因此，我们关注 H 的最小权顶点覆盖 [，参见例如 44]。顶点覆盖 $V$ 的权重为 $w(V) = \sum_{v \in V} w(v)$。最小权顶点覆盖 $V$ 是指其权重 $w(V)$ 为

[2]我们称它们为顶点，以便与属性图中的节点区分开来。

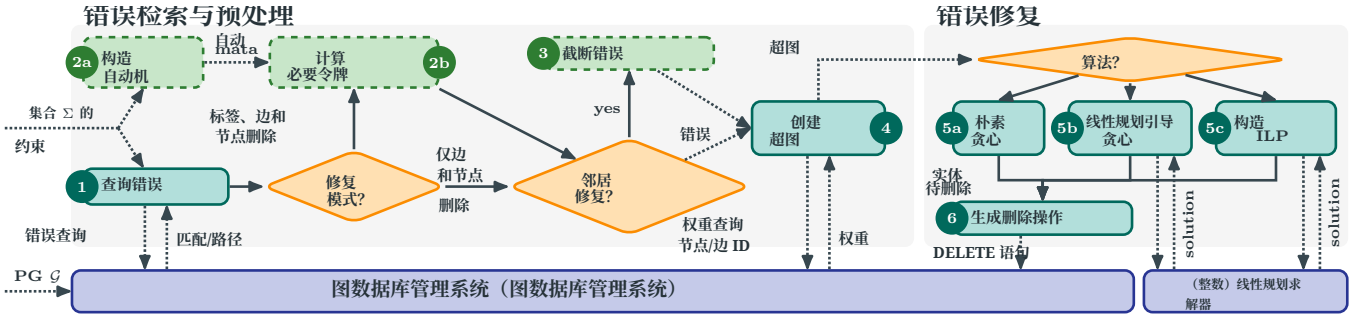



**图 2：我们的属性图修复流程 f or 由 6 个步骤组成，其中步骤 2 和 3 为可选步骤。2a 和 2b 可以是以下任一种 th 启用与否，而 5a、5b 和 5c 是其他选项。实线边表示控制流，虚线边表示通信 n.**

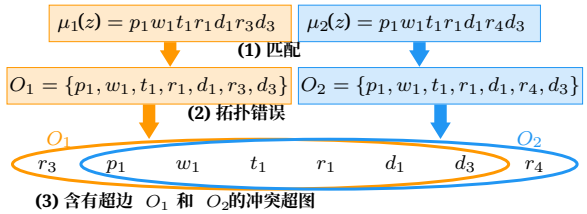



**图 3：拓扑错误的不同编码方式**

所有顶点覆盖（针对 $\mathcal{H}$）中最小的。注意，如果所有顶点的权重相同，那么最小权顶点覆盖与最小顶点覆盖相同[3]，；最小顶点覆盖的概念比极小顶点覆盖更强。每个最小顶点覆盖都是极小顶点覆盖（但反之不成立）。下面给出一个结果，其证明见附录 B.2。

命题 4.7。给定一个最小权顶点覆盖 $V$，从 $G$ 中移除[4] 的所有对象 $V$，即可得到 $G$ 的拓扑修复。

以下整数线性规划 (ILP) 为每个顶点 $v \in W$ 定义整数变量 $x_v$，用于编码寻找最小权顶点覆盖的问题。

$$\text{minimize} \sum_{v \in \mathcal{W}} w(v) \cdot x_v \quad \text{subject to} \sum_{v \in O} x_v \geq 1 \quad \text{for all } O \in \mathcal{E}$$
$$\text{and } x_v \in \{0,1\} \quad \text{for all } v \in \mathcal{W}$$

给定该 ILP 的一个解，即一个将所有变量 $x_v$ 映射到 {0，1} 的函数，它对应一个最小权顶点覆盖 $V$：该覆盖由所有满足 $x_v$ 映射为 1 的顶点 $v$ 构成。

总体而言，可以通过将每个约束重写为错误查询、查询错误，然后求解上述整数线性规划来计算修复。由于超边数量至多呈指数级，且整数线性规划属于 NP，整个过程的运行时间为非确定性指数时间（数据复杂度）。

贪心修复（步骤 *5a* 和 *5b*）。尽管整数线性规划求解器在实践中速度相当快，但有时我们仍希望以修复质量（在我们的情形中，即删除的对象数量）为代价，换取更短的运行时间。实现这一目标的常见方法是采用所谓的贪心算法，我们将讨论其中两种。

我们将第一种称为朴素贪心算法。它分两个阶段计算一个权重较小的顶点覆盖 $V_g$：选择阶段和修剪阶段。在选择阶段，算法遍历所有超边，选取顶点覆盖的候选顶点：如果当前超边包含一个已被选中且在该超边所有顶点中权重最小的顶点，则跳过该超边。否则，算法从该超边中任意选取一个权重最小的顶点。选出的顶点构成一个顶点覆盖$V_g$，因为每条超边都至少包含一个选出的顶点。

然而，该算法最终可能会选择过多的顶点，因此 $V_g$ 可能不是极小的。因此，这一阶段得到的是一个近似修复。为了获得修复，该算法会在修剪阶段尝试再次减少所选顶点的数量。它通过按权重降序排列每个所选顶点 $v$，并检查是否存在某条超边 $E$，使得 $v$ 是其中唯一被选中的顶点（即满足 $V_g \cap E = \{v\}$）。如果不存在这样的超边 $E$，则必须从 $V_g$中移除 $v$。

再次考虑这两个错误 $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ 和 $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$。朴素贪心算法可能会在第一阶段计算出 $V_g = \{r_3, r_4\}$。在第二阶段，$V_g$不会改变，因为 $r_3$和 $r_4$分别不出现在 $O_2$和 $O_1$中。因此，顶点覆盖 $V_g$ 是极小的，因为 $r_3$ 和 $r_4$ 都无法被移除。然而，它并非最小权顶点覆盖，因为例如，最小顶点覆盖 $\{r_1\}$ 的权重更小。[5]尽管朴素贪心算法不一定能计算出最小权顶点覆盖，但选择权重最小的顶点足以计算出一个（拓扑）修复。由于该算法会遍历所有错误，因此其运行时间为指数级。

命题 4.8。给定通过朴素贪心算法计算得出的一个极小顶点覆盖 $V_g$，从 $G$ 中移除 $V_g$ 中的所有对象，即可得到 $G$的拓扑修复。

证明见附录 B.2。

对于第二种贪心算法，我们考虑一种线性规划引导的贪心算法，它将 ILP 的*LP* 松弛 的解与朴素贪心算法相结合。这里，LP 松弛是指通过将 ILP 中的条件 $x_v \in \{0,1\}$替换为不等式 $0 \leq x_v \leq 1$，并允许$x_v$取实数值而得到的线性规划（LP）。众所周知，LP 可以求解

[3] $V$是一个最小顶点覆盖，当且仅当不存在满足 $|V'| < |V|$的顶点覆盖 $V'$。
[4] 回想一下，在 $G$中，移除一个节点意味着移除其所有关联边。
[5] 朴素贪心算法也可能选择 $r_1$，而不是 $r_3$ 和 $r_4$；在这种情况下，它会在多项式时间内得出最小权顶点覆盖 $\{r_1\}$。

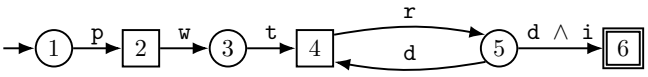



**图 4：示例 3.2中的 RGPC 模式自动机**

与整数线性规划不同，该算法能在多项式时间内运行。随后，算法选择任意一个满足 $x_v > 0$ 的顶点 $v$ 作为候选项，然后按照朴素贪心算法继续执行。

## 4.3 可选步骤 2：标签删除

我们继续介绍图 2中的可选步骤 2，该步骤属于我们的修复流程，可通过删除标签（以及节点和边）来获得非拓扑修复。

无论使用整数线性规划还是贪心算法，我们的修复流程都只会在错误中包含孤立节点（也称为长度为 0的路径）时删除节点。否则，由于边的权重最小，为 1，因此会删除边。如果删除某个节点的所有关联边，修复结果中就可能出现（数量可能很多的）孤立节点。为避免这种情况，我们研究不一定是拓扑修复的修复。更确切地说，我们将方法扩展为可通过移除标签来获得修复，如示例 4.1所述。与删除某个节点的（许多）关联边相比，移除该节点的单个标签就可能足够。请注意，仅移除标签并不总能获得修复，例如，当某个约束不涉及任何标签时。因此，除了删除标签外，可能仍有必要删除节点和/或边。

其思路是将拓扑错误 $O_\mu$扩展为包含对象对 $(o, \ell)$ $o$（即节点或边）和标签 $\ell$。最初，拓扑错误包含属性图中出现在匹配 $\mu$ 的陪域中的节点和边。我们将这些对象对 $(o, \ell)$ 称为标记。其含义是，可以通过移除节点（或边） $\ell$ 上的标签来修复错误 $o$。因此，（不一定是拓扑的）冲突超图的边，就是所有扩展后的集合——下文中我们将其简称为错误。需要注意的是，一般而言，一个拓扑错误可以有多个扩展。特别是，错误的数量多于拓扑错误。

下面我们讨论如何在给定匹配的情况下计算错误。我们从单个约束包含单个路径模式这一简单情况开始。再次考虑示例 3.2 中的约束。回想一下，它由一个包含路径变量 $z$的路径模式组成。我们可以观察到，从节点 $p_1$移除 person 标签，或者从边 $w_1$ 移除 works_on 标签，都能得到一个修复。事实上，与该约束的模式匹配的每条路径都必须以一个带有 person 标签的节点开头，后接一条带有 works_on 标签的边，而$p_1$ 和 $w_1$ 是唯一带有这些标签的对象。因此，($p_1$, person) 是我们要找的标记。($w_1$, works_on) 也是如此。我们称这些标记为必要标记。

**自动机构造（步骤 *2a*）。**为了自动计算任意 RGPC 路径模式的必要令牌，我们构造自动机，称之为 *RGPC* 自动机，并将在下文通过一个示例介绍。考虑路径模式

$$(x:\mathrm{p}) \xrightarrow{:\mathrm{w}} (u:\mathrm{t})\Big[\xrightarrow{:\mathrm{r}} (:\mathrm{d})\Big]^{+}(y:\mathrm{d} \wedge \mathrm{i})$$

示例 3.2中的路径模式对应的 RGPC 自动机如图 4所示。请注意，转移标记为标签表达式，即连接标签的命题公式，而不只是（单个）标签。例如，从状态 5 到状态 6 的转移标记为 d ∧ i，这是节点模式（$y$:d ∧ i）的标签表达式。因此，RGPC 自动机属于所谓的符号自动机 [，参见例如 14]。如果节点或边的标签集合满足该标签表达式，则可以执行相应的转移。例如，带有公式 d ∧ i 的转移实际上表示多个转移：每个包含 d 和 i 的标签集合对应一个转移。值得注意的是，当 d 和 i 同时存在时，也可以执行从状态 5 到状态 4 的转移。这使自动机能够读取路径的轨迹，例如路径 $p_1 w_1 t_1 r_1 d_1 r_3 d_3$ 第 2.1 节末尾的路径轨迹 {p}{w}{t}{r}{d, i}{r}{d, i}。

此外，与经典自动机不同，图 4 所示的 RGPC 自动机包含两种状态：一种画成圆形，另一种画成方形，分别表示下一次转移读取节点的标签或边的标签。例如，状态 1 的出边转移源自节点模式（: p），状态 2 的出边转移则源自边模式 $\xrightarrow{:\mathrm{w}}$。特别是，RGPC 自动机交替读取节点标签和边标签，因此能够读取一条路径的轨迹。

最后请注意，路径模式中的变量对自动机构造不起作用：我们使用 RGPC 自动机重新扫描错误检索步骤（步骤 1）返回的路径。因此，我们已经知道这类路径是匹配项，而我们只关心确定要删除的标签。

RGPC 自动机构造的形式化内容见附录 B.1。

**算法 1：使用必要令牌计算错误针对单条路径**

**输入：** 匹配 $\mu$，路径模式 $z = \mathrm{P}$，$\mu(z) = v_0 e_1 v_1 \ldots e_n v_n$
**输出：** 包含必要令牌的错误集合 $\mathrm{E}_\mu$

```
A ←P 的自动机;
O_μ ← {v_0, e_1, v_1, ..., e_n, v_n};                    // 拓扑错误
E_μ ← ∅;                                                  // 错误集合
对每个接受运行 r 于路径 A μ(z) 执行
    O_r ← ∅;
    对于每个 o ∈ O_μ 并且对标签 ℓ 执行
        如果 从 o 移除 ℓ 会使 r 无效则
            O_r ← O_r ∪ {(o, r)};                         // (o, ℓ) 是必要的
    O_{μ,r} ← O_μ ∪ O_r; E_μ ← E_μ ∪ {O_{μ,r}};
返回 E_μ
```

计算必要令牌（步骤 2b）。算法 1 概述了我们针对单个路径模式 P 计算错误的算法。该算法使用步骤 2a 中构造的自动机来识别必要令牌，方法如下：令 $z$ 为 P 的路径变量，A 为为其构造的自动机，$\mu$ 为 P 的一个匹配。则 A 接受路径 $\mu(z)$ 的轨迹。为获得必要令牌，我们考察 A 在 $\mu(z)$ 的轨迹上的每个接受运行 $r$：如果从路径 $\mu(z)$ 上的某个对象 $o$中移除一个标签$\ell$会使该运行失效，那么该标签对于此运行就是必要的。例如，轨迹 {p}{w}{t}{r}{d, i}{r}{d, i} 被图 4 所示的自动机接受。从轨迹中最后一个集合移除标签 i 会使所有接受运行失效，因为自动机将无法再执行从状态 5 到 6 的转移

在读取第二个 {r} 之后（如果更早执行该转移，自动机就无法读完整条轨迹）。

要使匹配 $\mu$ 失效，必须使所有接受运行都失效。因此，对于每个接受运行 $r$，我们通过添加对 $r$ 必要的所有标记，将 $O_\mu$ 扩展为错误 $O_{\mu,\ r}$。请注意，如果模式不包含任何标签，那么任何运行都没有必要令牌。在这种情况下，原始错误 $O_\mu$ 保持不变。

现在假设该约束由图模式 $q$ 及其 $k$ 个路径模式组成，路径变量为 $z_1$、……、$z_k$。设 $\mu$ 是一个见证匹配，表明该约束未得到满足。再设 $\mathrm{A}_i$ 是模式 $i \in \{1, \ldots\ldots, k\}$ 的自动机，$r_1$、……、$r_k$ 是 A1、……、$\mathrm{A}_k$ 在路径 $\mu(z_1)$……、$\mu(z_k)$ 的轨迹上的接受运行。虽然只需使其中一个运行失效，就能使组合 $r_1$、……、$r_k$ 失效，但我们注意到，可能还存在涉及相同运行的其他组合：假设我们通过删除某个标签使接受运行 $r_j$ 失效。那么，运行 $r_1$、……、$r_j$、……、$r_k$ 将不再能证明 $\mu$ 是一个匹配。然而，可能存在另一个运行 $r'_j$，使得 $r_1$、……、$r'_j$、……、$r_k$ 仍能证明 $\mu$ 是一个匹配。

因此，我们为每种接受运行的组合 $r_1, \ldots, r_k$ 创建一个错误。更准确地说，我们将集合 $O_{\mu,r_1}, \ldots, O_{\mu,r_k}$ 合并为一个错误 $O_{\mu,r_1,\ldots,r_k} = O_{\mu,r_1} \cup \cdots \cup O_{\mu,r_k}$。对于每个匹配 $\mu$ 以及所有运行组合 $r_1, \ldots, r_k$，集合 $O_{\mu,r_1,\ldots,r_k}$ 即为冲突超图的超边。这样，必须通过移除至少一个对象来使所有组合失效。

由于移除节点或边意味着移除其所有标签，因此我们相应地调整权重：对于一条边，其权重为其标签数量加 1；对于一个节点，其权重为其标签数量、其关联边的权重之和，再加 1（代表节点自身）。令牌 $(o, \ell)$ 代表单个标签，因此权重为 1。

我们强调，命题 4.7 并不适用，因为 RGPC 模式中的公式可能包含否定：因此，删除标签可能会产生新的匹配，进而导致新的错误。我们称一个 RGPC 约束为正向的，当其模式中的标签表达式均不包含任何否定时。由此可得以下结论。

命题 4.9。 如果给定的 *RGPC* 约束是正向的，则可在非确定性指数时间内（数据复杂度）计算通过删除标签进行的修复。

## 4.4 可选步骤 3：邻域错误

可选的步骤 3 试图通过限制错误的规模来缩短运行时间，从而也减少错误的数量。作为权衡，修复流程通常只能得到近似修复。

错误的规模确实取决于所考虑属性图的大小，而错误的数量可能呈指数级增长。我们通过引入邻域错误来解决这个问题：简言之，其思路是移除靠近导致错误的路径端点的对象，从而修复属性图。

对于整数 $k \geq 1$，可以直观地将其理解为路径端点周围的半径，路径的 $k$-端点邻域 由与端点 $v_0$ 和 $v_n$ 之一的距离至多为 $k$ 的对象组成，也就是说，这些对象是 $v_0$、$e_1$、$v_1$、……、$e_k$、$v_k$ 和 $v_{n-k}$、$e_{n+1-k}$、$v_{n+1-k}$、$\ldots e_n$、$v_n$。

给定一个 RGPC 约束 $q$；$F \Rightarrow C$ 其中包含 $\ell$ 个路径变量 $z_1$、……、$z_\ell$，以及一个匹配 $\mu$，它表明该约束未满足，则拓扑 $k$-邻域错误 $O^k_\mu$ 由以下部分组成：所有节点和边 $o$，对于它们，存在一个路径变量 $z_i, 1 \leq i \leq \ell$，使得 $o$ 出现在 $k$ 端点的邻域中，即 $\mu(z_i)$。

启用步骤 3 后，流水线会使用（拓扑）$k$-邻域错误来代替（拓扑）错误。例如，再次考虑示例 3.2 中的约束、图 1 所示的属性图，并回想一下，有两个匹配可以证明该属性图不满足此约束：第一个匹配对应于路径 $p_1 w_1 t_1 r_1 d_1 r_3 d_3$（参见示例 2.3），第二个对应于路径 $p_1 w_1 t_1 r_1 d_1 r_4 d_3$。拓扑 1-邻域错误为 $O^1_1 = \{p_1, w_1, t_1\} \cup \{d_1, r_3, d_3\}$，以及 $O^1_2 = \{p_1, w_1, t_1\} \cup \{d_1, r_4, d_3\}$。与（完整的）拓扑错误 $O_1 = \{p_1, w_1, t_1, r_1, d_1, r_3, d_3\}$ 和 $O_2 = \{p_1, w_1, t_1, r_1, d_1, r_4, d_3\}$ 相比，边 $r_1$ 在 $O^1_1$ 和 $O^1_2$ 中缺失。因此，使用 $O^1_1$ 和 $O^1_2$ 代替 $O_1$ 和 $O_2$ 时，不会找到通过删除 $r_1$ 获得的修复。另一方面，这些错误规模更小，而且一般来说，其规模的上界与 $k$ 呈线性关系，而非与属性图的规模呈线性关系。

## 5 实验

为展示我们的方法在实践中的适用性，我们开展了若干实验：首先，我们评估基础流水线在错误数量和约束数量方面的运行时间与可扩展性，并比较整数线性规划与两种贪心算法的性能和质量。接着，我们研究修复流程中的可选步骤对质量和运行时间的影响。最后，我们开展定性研究，考察该流水线在包含真实世界约束的真实世界数据集上的表现。评估中，我们使用表 1 中列出的四个不同规模的属性图。前三个是现实世界的属性图，而 LDBC 属性图则通过相应的基准测试生成。我们使用 Python 3.13 实现了图 [41], 2 所示的修复流程。求解（整数）线性规划问题时，我们使用 HiGHS [28]。作为图数据库系统，我们使用 Neo4j 5.26（社区版）。所有实验均在一台虚拟机上运行，该虚拟机配备 8 个 Intel(R) Xeon(R) Silver 4214 CPU 物理核心 [17] 和 125 GiB 内存。

### 5.1 评估基础流水线

在本节中，我们评估基础流水线的性能和质量，也就是不含任何可选步骤的修复流程。为此，我们针对表1中的属性图，按照图5 所示的模式形状生成了约束。这些模式形状涵盖了真实查询日志中观察到的大多数形状 [9]，同时也包含递归。值得注意的是，仅单向形状就涵盖了观察到的有效路径模式的 74.61% [9],；环形状则包括三角形，因为橙色虚线形状可以构成匹配长度为 2 的路径的模式。一个包含 3 个边模式和 4 个节点模式的单向模式实例为 $z = [(:\mathrm{a})_{:\mathrm{c}} \longrightarrow (:\mathrm{b})_{:\mathrm{e}} \longrightarrow (:\mathrm{d})_{:\mathrm{f}} \longrightarrow (:\mathrm{h})]^+$。随后，一个约束由一个模式、一个空筛选条件和条件 {false} 组成。这使扩展性实验中的错误数量达到最大。此外，为了增加约束（和错误）的数量，我们在每个图中随机添加了相当于原有边数量 10% 的边，并为这些边赋予一个特殊标签；它们连接的节点在原图中已由一条边相连，从而保持拓扑结构不变。我们强调，新增边与错误之间没有关联：它们可能参与错误，也可能不参与，和其他边一样。新增边的唯一目的是增加路径数量，并且

**表 1：用于实验的属性图、约束数量 $|\Sigma|$ 及错误数量**

| Name | #节点 | #边 | 单向 | | 两次重复 | | 双向 | | loop | | 三分 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | $\|\Sigma\|$ | 错误数 | $\|\Sigma\|$ | 错误数 | $\|\Sigma\|$ | 错误数 | $\|\Sigma\|$ | 错误数 | $\|\Sigma\|$ | 错误数 |
| ICIJ 属性图 [29] | 2 016 523 | 3 339 267 | 14 | 558799 | 19 | 397108 | 39 | 7553 | 188 | 56614 | 152 | 876933 |
| 意大利立法图 [15, 16] | 411 787 | 1 117 528 | 20 | 166977 | – | | 53 | 443695 | 61 | 3783 | – | |
| Coreutils代码属性图 [45] | 206 506 | 387 284 | 20 | 45728 | – | | 52 | 3707 | – | | – | |
| LDBC，缩放因子 0.3 [1, 2] | 940 322 | 5 003 201 | 5 | 79064 | 6 | 72756 | – | | 56 | 423 | 4 | 2734499 |

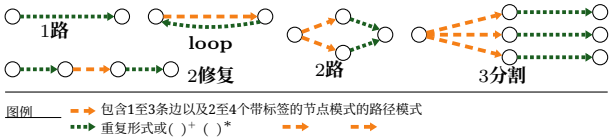



**图 5：第5.1 节和5.3节所用 RGPC 模式的形状**

因此也增加错误数量。表1列出了每种形状和属性图对应的约束数量及错误总数。请注意，我们没有真实数据。相反，我们将给定的图视为脏图，而我们的修复按定义是干净的。整数线性规划算法得到的修复被视为可能的最佳修复。

我们比较两种贪心算法和整数线性规划算法的性能与质量。我们以运行时间[6] 衡量性能，并以删除数量作为质量指标——删除数量越少，质量越高，因为这意味着信息损失越少。请注意，诸如用于衡量质量的 $F_1$-分数之类的指标并无意义：由于我们的整体方法总会修复所有错误，因此不存在假阴性；而根据定义，修复中也不可能包含假阳性。因此，$F_1$-分数始终为 1（超时的情况下为 0）。

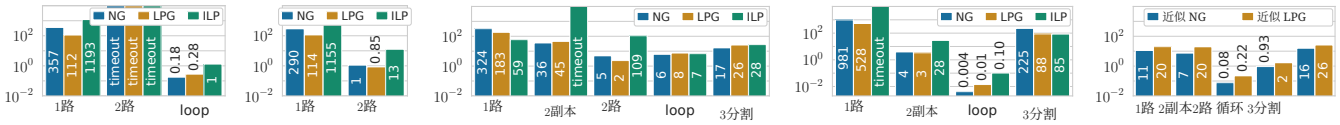
图 6a 至 6d 展示了朴素贪心算法（NG）、线性规划引导的贪心算法（LPG）和整数线性规划算法的运行时间。请注意，我们使用了对数刻度纵轴，以便绘图清晰易读。例如，图 6a展示了基础管道应用于立法属性图时的性能，该图采用表 1-路、2-路和环约束集合，详见表 1。总体而言，整数线性规划算法比朴素贪心算法耗时更长；作为权衡，它删除的对象更少[7]，参见图 7a 至 7d（这些图同样采用对数刻度纵轴，但范围与运行时间图不同）。在几乎所有情况下，线性规划引导的贪心算法都能达到与整数线性规划算法相同的质量（参见图 7a 至 7d）。与此同时，它在性能方面可与朴素贪心算法相媲美。在许多情况下，它甚至表现更好。检查线性规划求解器的解后发现，在这些情况下，解会逐渐趋近整数解。

总体而言，实验表明，线性规划引导的贪心算法所得到的修复质量更高，且删除数量比朴素贪心算法少约35%，修复质量与整数线性规划算法所得到的相当。与整数线性规划策略相比，它还具有超过97%的运行时间优势（图6c，2路约束），并且在整数线性规划算法超时的情况下胜过朴素贪心算法（图6d，1路约束）。然而，在一般情况下，这些观察结果并非总是成立，如图6c（ICIJ）所示：对于1路约束，整数线性规划算法优于两种贪心算法；这类约束中的错误数量较多，且路径包含许多环。

近似修复。现在，我们通过禁用流水线中的修剪阶段来研究近似修复，并进一步比较朴素贪心算法与线性规划引导的贪心算法。ICIJ 图的结果如图6e 和7e所示。我们观察到，朴素贪心算法的选择阶段表现优于线性规划引导的贪心算法的选择阶段；与图6c中的完整运行时间相比，修剪阶段占两种算法运行时间的大部分。具体而言，朴素贪心算法的修剪阶段使其整体表现较差。对建议删除的边数量进行比较后可以看出，这是因为朴素贪心算法有更多边删除被修剪掉。当线性规划引导的贪心算法有相对较多的边删除需要修剪时，朴素贪心算法可能胜过它（例如 ICIJ，2-rep）。最后，在大多数情况下，线性规划引导的贪心算法的选择阶段能够得到良好的近似结果，甚至获得正确修复，同时将运行时间缩短89%（1-way）。其他数据集的类似结果见附录C。

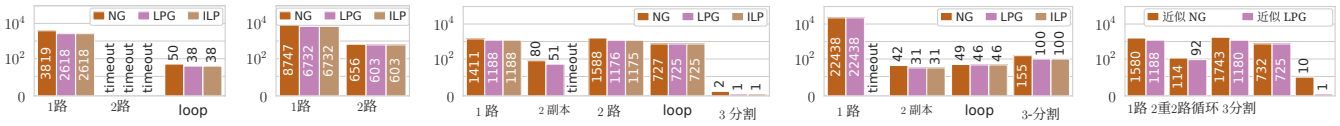
可扩展性。就流水线在约束数量和错误数量方面的可扩展性而言，图 6c 和图 6a 表明，对于 ICIJ 图，我们的流水线最多可处理 188 个环约束；而对于规模较小、包含 53 个双向约束的立法图，流水线则会超时。就错误规模而言，我们观察到的最大规模为 73 个对象（立法图，环约束）。图 6d 显示，超过 270 万个由三分约束导致的错误可以相对快速地修复，而由单向约束导致的 79064 个错误则颇具挑战性。比较 ICIJ 图和 LDBC 图的单向约束结果，我们发现，尽管规模较小的 LDBC 图中同形状的约束更少、错误也更少（参见表 1 的“单向”列），我们的流水线在前者上的表现仍然更好。为进一步研究这一情况，我们针对 ICIJ 图和 LDBC 图使用部分单向约束运行了实验，从而减少错误数量。运行时间结果如图 8a 和图 8b 所示，删除数量则如图 8c 和图 8d 所示。这些图中的 $y$ 轴标注的是错误数量。对于 LDBC 图，相对于错误数量，删除的边数较多——在线性规划引导的贪心算法和整数线性规划算法中最多达 67%，在朴素贪心算法中则达 73%——而修复 ICIJ 图只需删除极少量的边。这表明，如果修复只需少量删除操作，我们的修复流程就能表现良好。约束和错误数量的影响相对可以忽略不计。

[6]我们不通过查询来衡量错误检测，因为这在很大程度上取决于数据库系统，并非本文的重点。[7]由于约束的性质，流水线始终选择删除边，且未启用任何可选功能。



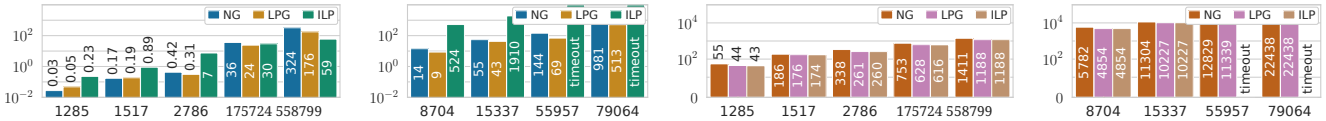

(a) 立法 (b) Coreutils (c) ICIJ (d) LDBC (e) ICIJ/近似

**图 6：朴素贪心算法（NG）、线性规划引导的贪心算法（LPG）和整数线性规划算法的运行时间（秒）**



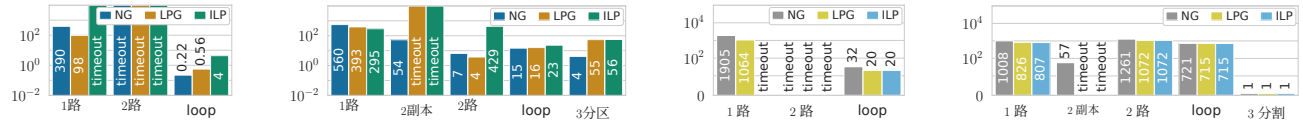

(a) 立法 (b) Coreutils (c) ICIJ (d) LDBC (e) ICIJ/近似

**图7：朴素贪心算法（NG）、线性规划引导的贪心算法（LPG）以及整数线性规划算法所提出的边删除数量**




(a) ICIJ/运行时间/1路 (b) LDBC/运行时间/1路 (c) ICIJ/边删除/1路 (d) LDBC/边删除/1路

**图8：朴素贪心算法（NG）、线性规划引导的贪心算法（LPG）和整数线性规划算法的扩展性实验结果**




(a) 立法/运行时间 (b) ICIJ/运行时间 (c) 立法/标签删除 (d) ICIJ/标签删除

**图 9：运行时间（单位：秒），启用步骤 2 后立法图和 ICIJ 图的标签删除数量** s

## 5.2 最小化重要对象的删除

默认情况下，我们的流水线为边赋予均匀权重，节点权重则由度决定。为提高鲁棒性，我们允许流水线为节点和边纳入不同的自定义权重 $w_c(o) > 0$，这些权重以属性图中的属性值形式存储 $o \in N \cup E$，旨在减少被视为“重要”的对象遭删除的情况。内部而言，边的权重 $w(e)$ 随后为$w_c(e)$；节点的权重 $w(n)$ 则为 $w_c(n)$ 加上所有 $w(e)$ 的总和，其中包含其所有关联边 $e$。因此，流水线的目标从最小化删除数量转变为最小化所有节点和边的总自定义权重的减少量。

为评估此功能，我们使用 PageRank 分数作为节点和边的权重[8]。不使用自定义权重时，在 ICIJ 图上应用 LP 引导的贪心算法后，PageRank 总值会下降 0。09% (1-向约束）。使用自定义权重时，下降幅度仅为 0。07% (2-向约束），对应的权重损失减少了 24%，这表明删除的高权重边更少。删除数量如图 10d所示，与图 7c相比，增幅范围为1%(2-重复约束）至43%(2-向约束）。运行时间如图 10c所示。与图 6c中的基准情况相比，自定义权重可能对运行时间产生正面或负面影响。更多详情见附录 C。

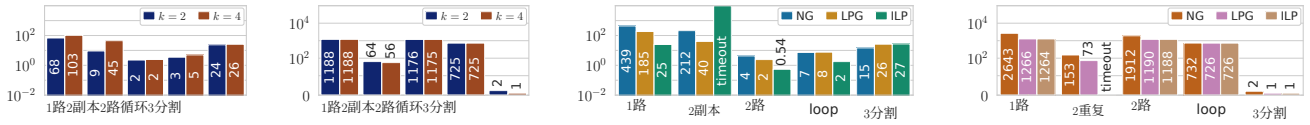
## 5.3 标签删除与邻域修复

现在，我们评估流水线中的可选步骤，并将其与第 5.1 节中确立的基础流水线进行比较。

为了展示步骤 2 的影响，我们在图 9 中展示了立法数据和 ICIJ 图的运行时间及标签删除数量。回想一下，用必要令牌扩展错误会导致错误更多、规模更大。因此，（整数）线性规划问题也会变得更大。此外，权重相同的修复往往也会更多：现在，除了删除一条边，还可以选择删除边上的任意标签或其某个端点。运行时间显著增加，最高可达 500%（图 6c 与图 9b，1 路）。整数线性规划算法还在另一个案例中超时（图 9a，1 路）。

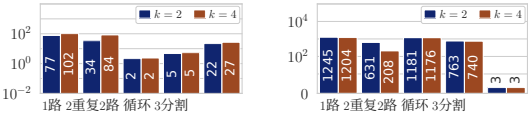
然而，与图7a 和图7c相比，我们观察到最多减少50%的对象删除（朴素贪心算法，立法，1-way），最多减少47%（整数线性规划算法，立法，循环），以及最多减少59%（线性规划引导的贪心算法，立法，1-way，此情况下整数线性规划算法超时，参见图9c）。

[8]边的 PageRank 使用线图计算：每条边都变成一个节点；如果原图中存在一条路径 $v_0 e_1 v_1 e_2 v_2$，则节点间存在一条边 $(e_1, e_2)$。




**(a) 步骤 3/运行时间** **(b) 步骤 3/边删除** **(c) 自定义权重/运行时间** **(d) 自定义权重/边删除**

**图 10：ICIJ 邻域修复（步骤 3）和自定义权重修复的线性规划引导的贪心算法结果**




**(a) ICIJ/运行时间** **(b) ICIJ/边删除**

**图 11：针对采样错误的线性规划引导的贪心算法**

值得注意的是，即使朴素贪心算法也能得到标签删除少于仅使用边删除的整数线性规划的修复（立法，1-way）。请注意，若允许标签删除，贪心算法有时甚至可以更快，这应归因于错误的结构以及（整数）线性规划（例如图9a，1-way）。

其他数据集也呈现出类似趋势；受篇幅所限，相关结果列于附录C。

采用步骤3 后，与基础流水线相比，运行时间最多缩短62 %，同时质量几乎保持不变，即删除数量基本相同。图中展示了 ICIJ 和邻域的运行时间及建议删除的边数量 $k = 2$ 以及 $k = 4$，针对线性规划引导的贪心算法，结果见图10a 和10b。与图6c所示的基准情况相比，当基础流水线耗时较长（即超过几秒）时，性能更好。

对于 $k = 2$ 和 $k$ =4，删除数量相当接近；更重要的是，除2-rep 约束的实验外，结果与图7c所示的基础流水线结果相同。也就是说，在所有匹配由最短路径组成且几乎完全被 $k$-端点邻域覆盖的情况下。

为了评估选取路径端点的邻域是否是减小错误规模的合适策略，我们使用错误的随机样本进行了相同的实验：也就是说，我们从每个错误中抽取了 $2k$ 条边（及其端点）。线性规划引导的贪心算法在 ICIJ 图上的结果如图 11所示。与图 10中所示的邻域错误结果相比，运行时间没有显著差异，但如果路径较长，删除数量可能会急剧增加（9.86x，适用于 2-rep）。这表明，（拓扑）错误的交集确实更有可能出现在路径端点附近，而从长路径中随机抽取样本则不太可能包含这些交集。

其他数据集的步骤 2 和3 对应的类似图表见附录 C。

## 5.4 定性研究

为了展示我们的修复流程在实践中切实可行，我们对真实世界的 ICIJ 属性图开展了一项定性研究，该属性图由国际调查记者同盟构建并使用。由于该属性图是从多个数据源组装而成，其中包含一些真实世界的不一致，这些不一致是使用表2中的 RGPC 约束检测到的。

这些约束涉及实体，它们对公司和组织进行建模；还涉及高管，即与实体有关的人员。值得注意的是，实体也可以处于与其他实体的高管_关系中，尽管其中任何一个都没有被标记为高管。约束 $\gamma_1$ 规定，处于相同_名称_关系中的实体和/或高管确实具有相同的名称。约束 $\gamma_2$ 是一条典型的否定约束：如果实体仍然有效，则禁止其具有失效日期。约束 $\gamma_3$ 对实体的排他性条件进行建模 [4]：任何实体都不能拥有两名唯一董事。最后，$\gamma_4$ 确保某个实体的总裁无法通过其总裁职务间接影响另一个与其有关联的实体，从而获得优势。每条约束对应的错误数量列于表 2 的第 3列中。

我们首先讨论基础流水线使用整数线性规划算法得出的修复结果。删除数量见表 2 第 4列。对于 $\gamma_1$，same_name_ 关系将被删除。这保留了属性中存储的原始名称（在此情形下，统一名称可能导致错误指控）。由于此约束对应的错误之间无法共享边，因此预期每个错误删除一条边。约束 $\gamma_2$ 就是一个示例：流水线只能删除节点，因为每个错误都恰好由一个节点组成。因此，这些节点会被删除，其所有 177 条关联边也会随之删除。对于 $\gamma_3$ 每个错误中的两条关系会删除其中一条，这表明对称性得到了妥善处理（在此情形下，错误之间不共享任何对象）。 $\gamma_4$ 的修复结果表明，删除数量可能显著少于错误数量——我们的流水线利用了错误之间存在共享边这一点。

我们使用 $\gamma_4$研究图拓扑变化下的鲁棒性，因为它包含一个非平凡的路径模式。假设描述由 same_name_as- 和 president_of- 标签边构成的路径的子模式很重要，我们将这些边的权重加倍。随后，修复流程提出删除 14 条边，而非 10 条，但保留了所有上述边。它还保留了 $\gamma_4$第一个模式的 203 个匹配中的至少 168 个。不使用自定义权重时，它仅保留该模式 96 至 117 个匹配。

启用标签删除后，图可以通过 6323 次标签删除得到修复，仅约为未启用时 18255 次节点和边删除的 35%。其中，1394 个标签从边上移除，这实际上删除了具体关系，但保留了存在某种（潜在）关系这一信息。

使用朴素贪心算法而非整数线性规划算法，导致 $\gamma_4$ 多删除了至多 2 项（具体取决于数据库返回错误的顺序），而对于其他约束，删除数量相同。线性规划引导的贪心算法产生的删除数量也相同。整数线性规划算法的运行时间不到 4 秒

**表 2：ICIJ 属性图的真实场景 RGPC 约束**

| ID | RGPC 约束 $q; F \Rightarrow C$ | 错误数 | 删除数 |
|---|---|---|---|
| $\gamma_1$ | $(x$:实体 $\vee$ 官员$)$ :同名 $\longrightarrow (y$: 实体 $\vee$ 官员$)$; $\emptyset \Rightarrow \{x.$名称 $= y.$名称$\}$<br>↪ 同一_名称_关系中的所有实体和高管确实具有相同的名称。 | 18000 | 18000 条边<br>0 个节点 |
| $\gamma_2$ | $(x$:实体$)$; $\{x.$失效_日期 $\leq$ 2025-07-01$\} \Rightarrow \{x.$状态 $\neq$ “有效”$\}$<br>↪所有失效日期已过的实体，状态都不是“有效”。 | 46 | 177 条边<br>46 个节点 |
| $\gamma_3$ | $(x$: 官员$)$ .唯一董事 $\longrightarrow (y$: 实体$)$, $(z$: 官员$)$ .唯一董事 $\longrightarrow (y)$; $\emptyset \Rightarrow \{x = z\}$<br>↪一个实体不能有多位唯一董事。 | 44 | 22 条边<br>0 节点 |
| $\gamma_4$ | $(x$: 官员$)$ : 总裁 $\longrightarrow (y$: 实体$)$ $[(:$ 实体 $\vee$ 官员$)$ : 任职于 $\xrightarrow[\vee]{}]^+$ 同名 $(z$: 实体$)$, $(x)$ : 任职于 $\longrightarrow (z)$; $\emptyset \Rightarrow \{y = z\}$<br>↪ 没有总裁通过空壳公司链等方式间接控制某个实体；他们直接在该实体中担任官员。 | 19 | 10 边<br>0 节点 |

对于所有约束的集合，在这种情况下，使用任一种贪心算法或邻域错误都没有优势。

## 6 相关工作

除了评估 PG约束 可被重写为用于查询错误的 GQL 查询 [4] 之外，针对 PG约束 的错误修复在先前工作中尚未得到探讨。

对表达能力较弱的图数据模型（如带标签图）的错误修复侧重于基于追赶的方案 [10, 11, 21, 22,32, 38, 39],，这些方案通过迭代添加属性甚至边来修复图。因此，这种修复模式与我们基于删除的方法正交，因而不可比较。特别是，仅插入的修复模式不适合修复我们示例和实验设置中否定约束的违反 [20]。

另一种修复图的方法是针对邻域约束，即根据节点邻居的标签来限制该节点标签的约束 [40]。他们的修复模式包括更改标签，若仍不足，则更改节点的邻域，这同样与我们的修复模式不兼容。他们引入了其算法的贪心变体，以在运行时间与修复质量之间取得平衡。

针对属性图的以用户为中心的修复已有研究 [36],，不过研究的是表达能力较弱的图数据模型所适用的约束。其修复模型侧重于标签和属性的更改，但也允许在不得已时删除边。让用户修复由变长路径构成的错误，需要精心设计人机界面，这超出了本文的研究范围。

使用冲突超图的顶点覆盖这一思路源自关系数据库修复的理论研究 [12, 42],，但这类修复不涉及对象间依赖和任意长的路径。一个值得注意的后果是，与关系场景不同，并非每个极小顶点覆盖都对应一种修复；我们通过使用最小加权顶点覆盖解决了这一问题。我们指出，与使用权重相比，显式建模依赖关系（如外键约束）会在属性图中产生数量不合理的（人为）错误，因为这些错误会涉及所有节点、边和标签。关于其他数据库模型和修复模式（包括关系模型）中错误修复的大量研究文献，请参阅综述 [13, 19]。

我们注意到，超图上的顶点覆盖问题甚至是W[2]-困难的 [25],，这意味着几乎不可能存在固定参数可处理算法。求解整数线性规划的一种常见策略是线性规划舍入算法 [，例如 35],，它们先求解整数线性规划的线性规划变体，然后使用随机舍入以获得整数解。我们的线性规划引导贪心算法则采用一个平凡的舍入步骤，这可以说是足够的，正如我们在第5节中的实验所示。

RGPC 模式的理论基础已得到研究，尤其是带路径变量的 CRPQ [6, 23],，它甚至允许对路径进行“正则”比较。值得注意的是，与 RGPC 模式不同，CRPQ 只能匹配边标签（即不包括节点标签），也不支持标签表达式。我们的 RGPC 自动机模型与为 CRPQ 构造的自动机也存在相同差异 [6]。

最后，我们将 RGPC 模式定义为 GPC 模式的一个语法片段，以抽象 GQL 和 SQL/PGQ 中模式匹配子语言的核心特征。[26]在实践中，一个实用且受这些语言支持的特性是允许沿反方向遍历边的模式。我们概述了如何为流水线扩展这一特性以及其他差异，详见附录 A.3。已知，对 GPC 模式进行匹配枚举所需的空间相对于属性图的大小是多项式空间，但相对于模式的大小则不是多项式空间 [26]；这使得在大型图中枚举所有错误变得难以处理，并为依赖枚举所有错误的修复过程设定了界限。最后，PG 约束能够表达基于正则路径查询的依赖关系，而相关的图修复问题（如确定查询应答）是不可判定的 [例如，参见 7, 27, 37]。

## 7 结论与未来工作

本文研究了 PG约束下属性图的错误修复。我们设计了一个针对 RGPC 约束的修复流程，并提供用户可选的权衡方案，还通过大量实验验证了其有效性。

未来，我们计划研究 PG约束下错误修复的实际局限。还可以纳入其他功能，例如高级 GQL模式匹配 [18, 24]。修复模型也可以扩展为允许属性删除，但这颇具挑战，因为删除可能引入新的错误。其他修复模式可以将我们基于删除的方法与最先进的插入技术结合起来。最后，鉴于递归约束可能产生大量错误，并受到关系场景中增量错误检测相关工作的启发 [31],，我们希望探索如何高效地从源头上防止错误被插入。

## 致谢

作者获得了 ANR-21-CE48-0015 VeriGraph 项目的资助。A. Bonifati 还获得了 IUF特聘讲席的资助。

# 引用


[1] Renzo Angles、János Benjamin Antal、Alex Averbuch、Peter Boncz、Orri Erling、Andrey Gubichev、Vlad Haprian、Moritz Kaufmann、Josep Lluís Larriba-Pey、Nor- bert Martínez-Bazan、József Marton、Marcus Paradies、Minh-Duc Pham、Arnau Prat-Pérez、Mirko Spasic、Benjamin A. Steer、Gábor Szárnyas 和 Jack Waudby。2020年。LDBC 社交网络基准测试。https://doi.org/10.48550/arxiv。2001.02299 arXiv:2001.02299[2] Renzo Angles、Peter A. Boncz、Josep Lluís Larriba-Pey、Irini Fundulaki、Thomas Neumann、Orri Erling、Peter Neubauer、Norbert Martínez-Bazan、Venelin Kotsev 和 Ioan Toma。2014年。关联数据基准委员会：图与 RDF 行业基准测试工作。ACM SIGMOD Record 43, 1（2014年5月），27–31。https://doi.org/10.1145/2627692.2627697[3] Renzo Angles、Angela Bonifati、Stefania Dumbrava、George Fletcher、Alastair Green、Jan Hidders、Bei Li、Leonid Libkin、Victor Marsault、Wim Martens、Filip Murlak、Stefan Plantikow、Ognjen Savkovic、Michael Schmidt、Juan Sequeda、Slawek Staworko、Dominik Tomaszuk、Hannes Voigt、Domagoj Vrgoc、Mingxi Wu 和 Dusan Zivkovic。2023年。PG-Schema：属性图的模式。ACM 数据管理会议录 1，2（2023年6月），1–25。https: //doi.org/ 10 1145 3589778 。/。Renzo Angles、[4] Angela Bonifati、Stefania Dumbrava、George Fletcher、Keith W. Hare、Jan Hidders、Victor E. Lee、Bei Li、Leonid Libkin、Wim Martens、Filip Murlak、Josh Perryman、Ognjen Savkovic、Michael Schmidt、Juan F. Sequeda、Slawek Staworko 和 Dominik Tomaszuk。2021年。PG-Keys：属性图的键。载于 SIGMOD '21：国际数据管理会议，线上会议，中国，2021年6月20日至25日，Guoliang Li、Zhanhuai Li、Stratos Idreos 和 Divesh Srivastava（编）。ACM，纽约，纽约州，美国，2423–2436。https://doi.org/ 10.1145/3448016.3457561[5] Marcelo Arenas、Leopoldo E. Bertossi 和 Jan Chomicki。1999年。不一致数据库中的一致查询答案。载于第十八届 ACM SIGACT-SIGMOD-SIGART 数据库系统原理研讨会论文集，1999年5月31日至6月2日，美国宾夕法尼亚州费城（SIGMOD/PODS99），Victor Vianu 和 Christos H. Papadimitriou（编）。计算机协会（ACM）出版社，纽约，纽约州，美国，68 79 10 1145 303976 303983 –。https://doi.org/。/。[6] Pablo Barceló、Leonid Libkin、Anthony W. Lin 和 Peter T. Wood。。基于图结构数据的路径查询表达语言。ACM 数据库系统汇刊 37，4（2012年12月），1–46。https://doi.org/10.1145/2389241。2389250[7] Catriel Beeri 和 Moshe Y. Vardi。1981年。数据依赖的蕴含问题。载于自动机、语言与编程，第八届专题讨论会，阿克（阿克科），以色列，1981年7月13日至17日，论文集（1981）（计算机科学讲义），Shimon Even 和 Oded Kariv（编），第115卷。施普林格柏林海德堡，柏林，海德堡，73 85 10 1007 3 540 10843 2_7 –。https://doi.org/。/---[8] Philip Bohannon、Michael Flaster、Wenfei Fan 和 Rajeev Rastogi。2005年。通过值修改修复约束的基于成本的模型与有效启发式方法。载于 ACM SIGMOD 国际数据管理会议论文集，美国马里兰州巴尔的摩，2005年6月14日至16日（SIGMOD-/PODS05），Fatma Özcan（编）。计算机协会（ACM），纽约，纽约州，美国，143 154 10 1145 1066157 1066175 –。https://doi.org/。/。[9] Angela Bonifati、Wim Martens 和 Thomas Timm。。大型 SPARQL 查询日志的分析研究。VLDB期刊 29，2–3（2020年5月），655–679。https://doi.org/ 10 1007 00778 019 00558 9 。/s---Yurong Cheng、Lei Chen、Ye Yuan 和 [10] Guoren Wang。2018 年。基于规则的图修复：语义化且高效的修复方法。载于第 34 届 IEEE 数据工程国际会议，ICDE 2018，法国巴黎，2018年4月16日至19日。IEEE 计算机学会，773 784 10 1109 2018 00075 –。https://doi.org/。/ [11] ICDE。。Yurong Cheng、Lei Chen、Ye Yuan、Guoren Wang、Boyang Li 和 Fusheng Jin。2022年。严格且灵活的基于规则的图修复。IEEE 知识与数据工程汇刊 34，7（2022年），3521–3535。https://doi.org/10。1109/TKDE.2020.3019817[12] Jan Chomicki 和 Jerzy Marcinkowski。2005年。使用元组删除进行最小变更完整性维护。Information and Computation 197，1–2（2005年2月），90 121 10 1016 2004 04 007 –。https://doi.org/。/J.IC。。Xu Chu、Ihab F. Ilyas、Sanjay Krishnan 和 Jiannan Wang。[13] 2016年。数据清洗：概述与新兴挑战。载于 2016 年国际数据管理会议论文集（2016-06）（SIGMOD/PODS'16）。计算机协会（ACM），旧金山加利福尼亚州美国，2201–2206。https://doi.org/ 10 1145 2882903 2912574 。/。Alessandro Cimatti、[14] Sergio Mover、Marco Roveri 和 Stefano Tonetta。2011年。从顺序扩展正则表达式到带符号标签的非确定有限自动机。载于自动机的实现与应用，Michael Domaratzki 和 Kai Salomaa（编）。第6482卷。施普林格柏林海德堡，柏林，海德堡，87–94。https://doi.org/ 10 1007 978 3 642 18098 9_10 。/----Andrea Colombo、[15] Anna Bernasconi 和 Stefano Ceri。2024年。意大利立法属性图。Zenodo。https://doi.org/ 10 5281 11210265 。/ [16] zenodo。Andrea Colombo、Anna Bernasconi 和 Stefano Ceri。。2024将立法系统建模为属性图以实现高级模式检测。https: //doi.org/10.48550/ARXIV.2406.14935 arXiv:2406.14935

[17] 英特尔公司. 2026年. 英特尔(R) 至强(R) Silver 4214处理器规格说明. https://www.intel.com/content/www/us/en/products/sku/193385/intel-xeon-silver-4214-processor-16-5m-cache-2-20-ghz/specifications.html. 访问日期: 2026-01-26.[18] Alin Deutsch、Nadime Francis、Alastair Green、Keith Hare、Bei Li、Leonid Libkin、Tobias Lindaaker、Victor Marsault、Wim Martens、J an Michels、Filip Murlak、Stefan Plantikow、Petra Selmer、Oskar Van Rest、Hannes Voigt、Domagoj Vrgoc、Mingxi Wu与Fred Zemke. 2022年. GQL与SQL/PGQ中的图模式匹配. 见《2022年国际数据管理会议论文集》（2022-06）（SIGMOD/PODS '22）. 计算机协会（ACM）, 美国宾夕法尼亚州费城, 2246–2258. https://doi.org/10.1145/3514221. 3526057[19] Ahmed K. Elmagarmid、Panagiotis G. Ipeirotis与Vassilios S. Verykios. 2007年. 重复记录检测：综述. IEEE 知识与数据工程汇刊 19, 1 2007 1 16 10 1109 2007 250581 （1月）, –. https://doi.org/ . /TKDE. . Wenfei Fan. . 重访依赖关系以改进数据质量. 见第二十七届 ACM SIGMOD-SIGACT-SIGART 数据库系统原理研讨会论文集, PODS 2008, 2008年6月9日至11日, 加拿大不列颠哥伦比亚省温哥华（SIGMOD/PODS '08）, Maurizio Lenzerini与Domenico Lembo（编）. 计算机协会（ACM）, 美国纽约州纽约, 159–170. https://doi.org/ 10 1145 1376916 1376940 . / . Wenfei [20] 2008 Fan、Wenzhi Fu、Ruochun Jin、Muyang Liu、Ping Lu与Chao Tian. 2023年. 使捕获图中的重复项与冲突变得可处理. ACM 数据管理会议录 1, 1（2023年5月）, 1–28. https://doi.org/10.1145/ 3588940[22] Wenfei Fan、Ping Lu、Chao Tian与Jingren Zhou. 2019年. 推导图的确定性修复. VLDB基金会论文集 12, 7（2019年3月）, 752–765. https://doi.org/ 10 14778 3317315 3317318 . / . Diego Figueira与 [21] Varun Ramanathan. 2022年. 扩展CRPQ的求值何时可处理? . 见PODS '22：国际数据管理会议, 美国宾夕法尼亚州费城, 2022年6月12日至17日（SIGMOD/PODS '22）, Leonid Libkin与Pablo Barceló（编）. 计算机协会, 美国纽约州纽约, 203 212 10 1145 3517804 3524167 – . https://doi.org/ . / . Diego [24] 2023 Figueira与Miguel Romero. . 单射语义下的合取正则路径查询. 见第42届ACM SIGMOD-SIGACT-SIGAI数据库系统原理研讨会论文集（2023-06）（SIGMOD/PODS '23）. 计算机协会（ACM）, 美国华盛顿州西雅图, 231–240. https://doi.org/ 10 1145 3584372 3588664 . / . Jörg Flum与Martin Grohe. . 参数化复杂 [25] 2006 性理论. 施普林格柏林海德堡, 柏林, 海德堡. https://doi.org/10.1007/3-540-29953-X[26] Nadime Francis、Amélie Gheerbrant、Paolo Guagliardo、Leonid Libkin、Victor Marsault、Wim Martens、Filip Murlak、Liat Peterfreund、Alexandra Rogova与Domagoj Vrgoc. 2023年. GPC：面向属性图的模式演算. 见第42届ACM SIGMOD-SIGACT-SIGAI数据库系统原理研讨会论文集（2023-06）（SIGMOD/PODS '23）. 计算机协会（ACM）, 美国华盛顿州西雅图, 241 250 10 1145 3584372 3588662 – . [27] 2017 https://doi.org/ . / . Nadime Francis与Leonid Libkin. . 数据图的模式映射. 见第36届ACM SIGMOD-SIGACT-SIGAI数据库系统原理研讨会论文集（2017-05）（SIGMOD/PODS'17）. 计算机协会（ACM）, 美国伊利诺伊州芝加哥, 389–401. https://doi.org/10.1145/ 3034786.3056113[28] Qi Huangfu与J. A. J. Hall. 2018年. 对偶修正单纯形法的并行化. 数学规划计算 10, 1（2018年12月）, 119–142. https://doi.org/10.1007/S12532-017-0130-5 https://highs.dev/. [29] 国际调查记者同盟. 2026年. ICIJ离岸泄密数据库. https://offshoreleaks.icij.org/pages/database. 访问日期: 2026-01-26. [30] ISO中央秘书处. 2024年. 信息技术——数据库语言——GQL（第1版）. 标准ISO/IEC 39075:2024. 国际标准化组织, 瑞士日内瓦. [31] Youri Kaminsky、Eduardo H. M. Pena与Felix Naumann. 2024年. 否定约束违例的增量检测. VLDB基金会论文集 18, 4 2024 1000 1012 10 14778 3717755 3717761 （12月）, –. [32] https://doi.org/ . / . Selasi Kwashie、Lin Liu、Jixue Liu、Markus Stumptner、Jiuyong Li与Lujing Yang. 2019年. Certus：一种采用图差分依赖（GDD）的有效实体解析方法. VLDB基金会论文集 12, 6（2019年2月）, 653–666. https://doi.org/10.14778/3311880.3311883[33] Yuxi Liu、Fangzhu Shen、Kushagra Ghosh、Amir Gilad、Benny Kimelfeld与Sudeepa Roy. 2024年. 子集修复表示的代价. VLDB基金会论文集 18, 2（2024年10月）, 475–487. https://doi.org/10.14778/ 3705829.3705860[34] Andrei Lopatenko与Leopoldo E. Bertossi. 2007年. 基数修复语义与增量修复语义下数据库中一致性查询回答的复杂性. 见《数据库理论——ICDT 2007，第11届国际会议，西班牙巴塞罗那，2007年1月10日至12日，论文集》（2006）（计算机科学讲义）, Thomas Schwentick与Dan Suciu（编）, 第 4353 卷 . 施普林格柏林海德堡 , 柏林 , 海德堡 , 179–193. https://doi.org/10. 1007/11965893


_13

[35] Mourad El Ouali、Helena Fohlin与Anand Srivastav. 2014年. 击中集问题的随机化近似算法. 理论计算机科学
555 2014 23 34 10 1016 2014 03 029 （10月）, – .
[36] 2025 https://doi.org/ . /J.TCS. . . Amedeo Pachera、Angela Bonifati与Andrea Mauri. . 以用户为中心的属性图修复. ACM 数据管理会议录 3, 1（2月
2025 85 1 85 27 10 1145 3709735 ）, : – : . https://doi.org/ . / Sylvain
[37] 2024 Salvati与Sophie Tison. . 路径约束下正则路径查询的包含, 见第27届国际数据库理论会议, ICDT 2024, 意大利帕埃斯图姆, 2024年3月25日至28日（2024）, Graham Cormode与Michael Shekelyan（编）. LIPIcs, 第290卷, ICDT 2024 290, 17:1–17:19. https: //doi.org/
10 4230 2024 17 . /LIPICS.ICDT. . Larissa Capobianco
[38] Shimomura、George Fletcher与Nikolay Yakovets. 2020年. 图生成依赖：图生成依赖. 见CIKM '20：第29届ACM国际信息与知识管理会议, 线上会议, 爱尔兰, 2020年10月19日至23日（CIKM '20）, Mathieu d'Aquin、Stefan Dietze、Claudia Hauff、Edward Curry与Philippe Cudré-Mauroux（编）. 计算机协会, 美国纽约州纽约, 2217–2220. https://doi.org/10.1145/3340531. 3412149[39] Larissa Capobianco Shimomura、Nikolay Yakovets与George Fletcher. 2024年. 基于图生成依赖对属性图进行推理. 信息科学
672 2024 120675 10 1016 2024 120675 （）, .
[40] https://doi.org/ . /J.INS. . Shaoxu Song、Boge Liu、Hong Cheng、Jeffrey Xu Yu与Lei Chen. 2017年. 邻域约束下的图修复. VLDB期刊 26, 5（2017年5月）,
611 635 10 1007 00778 017 0466 5 – . https://doi.org/ . /S - - -
[41] Christopher Spinrath、Angela Bonifati与Rachid Echahed. 2026年. PG约束下的属性图修复 *[制品]*. Zenodo. https://doi.org/10. 5281/zenodo.18301604[42] Sławek Staworko、Jan Chomicki与Jerzy Marcinkowski. 2012年. 关系数据库中的优先级修复与一致性查询回答. 数学与人工智能年刊 64, 2–3（2012年3月）, 209–246. https: //doi.org/
10 1007 10472 012 9288 8 . /s - - - Jef Wijsen. . 使用更新进行数据库修复. AC
[43] 2005 M 数据库系统汇刊 30, 3（2005年9月）, 722–768. https://doi.org/10.1145/1093382. 1093385[44] Mingyu Xiao、Sen Huang与Xiaoyu Chen. 2024年. 最大加权独立集：稀疏图上的有效归约与快速算法. Algorithmica 86, 5（2024年5月）, 1293–1334. https://doi.org/10.1007/s00453-023-01197-x[45] Fabian Yamaguchi、Nico Golde、Daniel Arp与Konrad Rieck. 2014年. 利用代码属性图建模并发现漏洞. 见2014年IEEE安全与隐私研讨会（2014-05）. IEEE, 加利福尼亚州圣何塞, 590–604. https://doi.org/10.1109/SP.2014.44

# 附录

本附录分为三个部分。在附录 appendix:rgpc 中，我们补充介绍 RGPC 模式和约束：与第 6 节相比，我们更详细地比较 RGPC 模式、CRPQ 和完整 GPC；进一步说明如何将 RGPC 约束写成 PG 约束；给出谓词的形式语义；并介绍一种通过反向遍历边进行扩展的可能方案。附录 appendix:repairing 补充介绍了 RGPC 自动机、第 4 节中缺失的证明，以及整数线性规划算法的一个替代版本。最后，在附录 C 中，感兴趣的读者可以找到更多实验结果，以及所有实验的原始测量数据和权衡百分比。

## RGPC 和 RGPC 约束

在附录 A.1 和 A.4 中，我们分别比较 RGPC 与 CRPQ 和完整 GPC。在附录 A.2 中，我们给出 RGPC 约束中谓词的形式定义。在附录A.3 中，我们说明如何通过反向遍历扩展 RGPC。

## A.1 与 CRPQ 的关系

为奠定理论基础，将 RGPC 模式视为（完整）合取正则路径查询（*CRPQ*） [的扩展可能会有所帮助，参见例如 6]。完整 CRPQ 由一组三元组 $(x, r, y)$ 构成，其中 $x$和 $y$是节点变量，而 $r$是一个定义在边标签上的正则表达式。在（属性）图中，三元组 $(x, r, y)$ 的一个匹配 $\mu$是一个映射，它将 $x, y$映射到图中的节点 $n, m$，并且在 $n$和 $m$之间存在一条路径，使得该路径的边标签 构成正则表达式 $r$所定义语言中的一个词。

例如，$(x, a(bb)^*, y)$ 匹配所有节点对 $v_0, v_n$，使得它们之间存在一条奇数长度的路径 $v_0e_1v_1 \dots e_nv_n$，其中 $e_1$标记为 $a$，而 $e_2, \dots e_n$标记为 $b$。它可以很容易地转换为 RGPC路径模式，即 $(x) : \mathtt{a} \longrightarrow \left[ : \mathtt{b} \longrightarrow : \mathtt{b} \longrightarrow \right]^*(y)$。其他三元组 $(x, r, y)$ 也可以用类似方式转换；因此，由多个三元组组成的 CRPQ 可以转换为 RGPC 图模式。我们注意到，所得 RGPC 模式不 包含任何节点标签，因为正则表达式 $r$ 只能引用边标签。此外，在边模式：$\xrightarrow{\varphi}$中，标签表达式 $\varphi$始终是单个标签（不包含任何合取、析取或否定）。节点变量只出现在路径模式的开头和结尾。

## A.2 谓词的形式语义

如第 3 节所述，RGPC 约束的筛选条件和条件中可能出现的谓词，其语义与通常情况相同，此外还要求谓词中出现的所有属性都必须存在。为完整起见，我们在下文给出形式化定义。回顾一下，$\oplus$取自 $\{=, \neq, \leq, \geq\}$ 中的元素。属性图 $G = (N, E, \rho, \lambda, \pi)$ 中，RGPC 模式 $q$ 的匹配 $\mu$ 满足谓词 $\ell$，当且仅当

- $\ell$的形式为 $x_i = y_j$ 且 $\mu(x_i) = \mu(y_j)$；
- $\ell$ 的形式为 $x_i \neq y_j$ 且 $\mu(x_i) \neq \mu(y_j)$；
- $\ell$的形式为 $x_i$.propA $\oplus$ $y_j$.propB，$\pi(\mu(x_i),$ propA) 和$\pi(\mu($ $y_j),$ propB) 均已定义，且

$$\pi(\mu(x_i), \text{ propA}) \oplus \pi(\mu(y_j), \text{ propB})$$

成立；

- $\ell$的形式为 $x_i$.propA $\oplus c$， $\pi(\mu(x_i),$ propA) 已定义，且$\pi(\mu(x_i),$ propA) $\oplus c$成立。

## A.3 添加反向遍历

RGPC 模式（以及我们的修复流程）可以通过反向遍历进行扩展，即能够沿反方向遍历边。为此，还必须扩展路径的定义，以允许边沿反方向延伸。

定义 *A.1*（路径）。设 $G = (N,$ $E,$ $\rho,$ $\lambda,$ $\pi)$ 是一个属性图。$G$中的一条路径 是一个非空交替序列 $v_0e_1v_1 \dots e_nv_n$，其中 $v_0,$ ……，$v_n \in N$ 是节点， $e_1,$ ……， $e_n \in E$ 是边，并且对于每个 $i \in \{1,$ ……，$n\}$， $\rho(e_i) = (v_{i-1},$ $v_i)$ 或者 $\rho(e_i) = (v_i,$ $v_{i-1})$ 成立。

现在可以通过反向边模式：$\xleftarrow{\varphi}$扩展 RGPC 的定义。与（正向）边模式：$\xrightarrow{\varphi}$不同，反向边模式要求在匹配路径时沿反方向遍历一条边。例如，$: \mathtt{a} \longleftarrow \cup : \mathtt{c} \longrightarrow : \mathtt{c} \longrightarrow$ 表示存在一条标签为 a 的反向边，或者 存在一条长度为二且两条边的标签均为 c 的路径。

## A.4 RGPC 与完整 GPC

在第3 节，我们将 RGPC 模式定义为 GPC模式的一个子集。正如文献 [26] 中所定义的那样，GPC 具有更多特性。值得注意的是，它还支持边变量，并允许变量出现在并集和重复运算符之下。这实际上又产生了另外两类变量，即可选变量和组变量，而且语义变得更加复杂——文献中使用类型系统来判定一个模式是否有效，并定义其语义 [26,第 4]节。对我们的片段而言，这并非必需。另一项特性是用于匹配属性值的条件。由于属性值已由我们约束的其他组成部分所涵盖，即 $P$和 $C$，因此我们不需要这一特性。GPC 还允许沿反方向遍历边，或遍历无向边。最后，路径模式配备了诸如“简单”和“最短”之类的限制符，它们将匹配限制为简单路径（不重复节点）和最短路径。限制符的目的是确保只存在有限多个匹配。若模式用于定义查询，这一点十分重要，但约束没有返回值。我们注意到，限制符也已针对 CRPQ 得到研究 [24]。

最后，我们允许使用命题公式来指定标签约束 $\varphi$。原始定义并不支持这一点[26]。虽然可以用连接和并集来表示标签的合取与析取，但严格来说，在 GPC 的原始定义中无法表示标签不存在的情况（即使用否定）。

## A.5 RGPC 约束与 PG约束的比较

正如引言中所述，RGPC 约束是 PG约束的一个子集。PG约束被提出作为具体图查询语言（尤其是 GQL）的扩展，并支持约束 [3, 4]。

在第 3 节中，我们简要讨论了示例 3.2 中的约束等价于图 1 所示的 PG 约束，该约束使用了 MANDATORY 关键字。下面我们将提供更多细节。

一般而言，规定必要元素的*PG* 约束 写作

对 $\vec{x}$ $p(\vec{x})$必须 $\vec{y}$ $q(\vec{x},\vec{y})$。

这里，$p(\vec{x})$ 和 $q(\vec{x},\vec{y})$ 分别是称为作用域 和描述符查询的查询。它们引用变量 $\vec{x}=(x_1,\ldots,x_n)$ 和 $\vec{y}=(y_1,\ldots,y_m)$，这些变量被赋给节点、路径或属性。

例 *A.2.* 图1所示的 PG 约束对应于例3.2中的 RGPC 约束，它使用类似 GQL 的作用域查询和描述符查询。MATCH 子句和第一个FILTER 子句构成作用域查询，其自由变量为x 和y。MATCH子句本质上是例3.2中 RGPC 图模式的 ASCII 艺术转换。第一个 FILTER 子句通过比较开始日期与当前时间，断言项目已经启动。

随后，MANDATORY 子句指出，对于MATCH子句模式的每一种可能匹配，赋给x 和y 的节点都必须具有 access_level 属性（更准确地说，这两个节点的记录都定义了键 access_level）。

最后，同样"必须"满足描述符查询；在本例中，它由第二个 FILTER 子句构成。

示例 A.2 中的翻译可以推广到任意约束 $q;F\Rightarrow C$：每个 RGPC 模式 $q$ 都可以用 GQL [26] 表示，而 $C$和 $F$则可以写成 FILTER 子句。

## B 第 4 节补充材料

附录 B.1 给出了我们对 RGPC 模式的自动机模型的形式化定义和构造；该模型在附录 4.3中进行了非正式说明。第 4.1 节的证明见附录小节 error-repairing-proofs。在附录 B.3 中，我们简要讨论了第 4.2 节所介绍的整数线性规划的一种替代方案（其效果较差）。

### B.1 RGPC模式的自动机

修复流程的步骤 2 依赖于自动机构造，下面将对此进行详细说明。我们将不含节点变量的 RGPC 路径模式 P 转换为自动机 A，使其恰好接受那些作为 P 匹配路径轨迹的词串。为便于理解和阅读，下面假设 P 不包含附录 A.3中介绍的任何反向遍历，即形如 $:\overleftarrow{\varphi}$ 的（子）模式。我们将在本节末尾讨论反向遍历。

我们首先定义自动机模型，其定义遵循经典符号自动机的定义，唯一的区别是状态分为两个集合 $Q_N$和 $Q_E$。来自 $Q_N$的状态表示下一次转移读取节点的标签，而来自 $Q_E$的状态表示下一次转移读取边的标签。我们用 $\mathrm{B}(S)$ 表示在命题集合 $S$ 上定义的命题公式集合。

定义 *B.1*（*RGPC* 自动机）。A（有限）*RGPC* 自动机$\mathcal{A}$ =（$Q_N$、 $Q_E$、L $q_0$、 Δ、 $F$）由以下部分组成

- 两个不相交的有限集合 $Q_N$、 $Q_E$，其中的元素为状态,
- 一个有限的标签集合L,
- 一个初始状态 $q_0\in Q_N$,

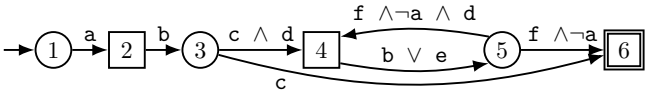



**图 12：示例 B.2 中 RGPC 模式对应的自动机B.2。**

- a finite *transition relation*

$$\Delta\subseteq(Q_N\times\mathcal{B}(\mathrm{L})\times Q_E)\cup(Q_E\times\mathcal{B}(\mathrm{L})\times Q_N),$$

and

- a set $F\subseteq Q_E$ of *accepting states.*

RGPC 自动机读取字母表 $\Gamma=2^{\mathrm{L}}$ 上的词串。A 的运行是标签集序列 $L_1\ldots L_n\in\Gamma^*$ 上的状态序列 $q_0q_1\ldots q_n$，以初始状态 $q_0$ 开始，并且对于每个 $i\in\{1,\ldots,n\}$，存在一个公式 $\varphi_i$，使得 $(q_{i-1},\varphi_i,q_i)\in\Delta$，且 $L_i\models\varphi_i$。这里，如果将 $L_i$ 中的所有标签解释为值为真的变量，并将其他所有标签解释为值为假的变量时，公式 $\varphi_i$ 的求值结果为真，则 $L_i\models\varphi_i$ 成立。若 $q_n\in F$，则运行 $q_0q_1\ldots q_n$ 为接受的。如果某个标签集序列 $L_1\ldots L_n$ 上存在接受运行，则 RGPC 自动机接受该序列；如果它接受属性图的一条路径 $p=v_0e_1v_1\ldots e_nv_n$的轨迹 $\lambda(p)=\lambda(v_0)\lambda(e_1)\lambda(v_1)\ldots\lambda(e_n)\lambda(v_n)$ $p$，则称它接受该路径。

*RGPC*路径模式的自动机 $z$ = P 是一个 RGPC 自动机 A，满足：对于每个属性图 $G$，A 接受路径 $p$ 当且仅当存在一个匹配 $\mu$，使得 $\mu(z)=p$，即 $p$ 被 $z$ = P 匹配。

例 *B.2*。 考虑路径模式

$$(:\mathrm{a})\xrightarrow{:\mathrm{b}}(:\mathrm{c})\big[(:\mathrm{d})\xrightarrow{:\mathrm{b}\ \vee\ \mathrm{e}}(:\mathrm{f}\ \wedge\ \neg\mathrm{a})\big]^*.$$

该路径模式对应的自动机如图所示12。

首先请注意，转移标记的是命题公式，而不是属性图的（单个）标签。例如，带有公式 f ∧ ¬a 的转移实际上代表多个转移：具体而言，每个包含标签 f 但不包含标签 a 的标签集合都对应一个转移。一般来说，显式计算（并写出）这些转移会导致自动机规模呈指数级增长。

例如，状态 1 的出边起始于节点模式 (:a)，状态 2 的出边起始于边模式 :b ⟶。

现在，我们可以将 RGPC 模式 P 转换为对应的 RGPC 自动机 A。为此，我们调整将正则表达式转换为自动机的标准构造方法：回顾一下，自动机交替读取节点和边上的标签；如果连续出现节点模式，且中间没有边模式，那么这些模式匹配的是同一个节点（标签）。我们注意到，RGPC 路径模式具有归纳结构：基本情形是节点模式 $(:\varphi)$ 和边模式 $:\xrightarrow{\psi}$，而复合模式则通过并集、重复和连接运算符组合路径模式构造而成。因此，我们可以根据 RGPC 路径模式 P 的结构归纳定义 A。在构造过程中，我们保持以下不变式：A 恰有一个接受状态 $q_f$，且接受状态$q_f$不是 初始状态 $q_0$。此外， $q_0$ 只有出边，$q_f$只有入边。最后需要强调的是，我们对符号自动机的定义不 允许 $\varepsilon$-转移。

设 L 为模式 P 中出现的标签集合。基本情形可按如下方式转换。

- P $= (:\varphi)$。节点模式断言存在一个标签满足 $\varphi$ 的节点。因此，我们定义

$$\mathcal{A} = (\{q_0\}, \{q_f\}, \mathrm{L}, q_0, \{(q_0, \varphi, q_f)\}, \{q_f\}).$$

- P $= :\xrightarrow{\varphi}$。除了断言存在一条边之外，P 还断言存在两个节点，即该边的两个端点。因此，我们定义

$$\mathcal{A} = (\{q_0, q_t\}, \{q_s, q_f\}, S, q_0, \{(q_0, \top, q_s), (q_s, \varphi, q_t), (q_t, \top, q_f)\}, \{q_f\}).$$

这里，$\top$ 表示重言式，它对每个标签集都成立。

为了构造复合路径模式 P，设 $\mathrm{A}^1 = (Q_N^1, Q_E^1, \mathrm{L}\ q_0^1, \Delta^1, \{q_f^1\})$ 和 $\mathrm{A}^2 = (Q_N^2, Q_E^2, \mathrm{L}\ q_0^2, \Delta^2, \{q_f^2\})$ 分别为针对 RGPC 路径模式 $\mathrm{P}_1$ 和 $\mathrm{P}_2$，构造的自动机。不失一般性，我们假设状态集 $Q_N^1, Q_E^1, Q_N^2, Q_E^2$ 两两不相交。我们为由 $\mathrm{P}_1$ 和 $\mathrm{P}_2$ 复合而成的 P 构造一个自动机 $\mathrm{A} = (Q_N, Q_E, \mathrm{L}\ q_0, \Delta, \{q_f\})$。

- P $= \mathrm{P}_1\mathrm{P}_2$。回想一下，根据两条路径连接的定义，$\mathrm{P}_1$ 匹配的最后一个节点对应于 $\mathrm{P}_2$ 匹配的第一个节点。因此，我们将 $q_f^1$ 的入转移与 $q_0^2$ 的出转移“合并”。

$$Q_N = Q_N^1 \cup Q_N^2, \quad Q_E = Q_E^1 \cup Q_E^2, \quad q_0 = q_0^1, \quad q_f^2 = q_f^2,$$
$$\Delta = \Delta^1 \cup \Delta^2 \cup \{(q^1, \varphi \wedge \psi, q^2) \mid (q^1, \varphi, q_f^1) \in \Delta^1, (q_0^2, \psi, q^2) \in \Delta^2\}$$

- P $= \mathrm{P}_1 \cup \mathrm{P}_2$。由于初始状态只有出转移，而接受状态只有入转移，因此只需将这些转移复制到新的初始状态 $q_0$ 和新的接受状态 $q_f$ 即可。

$$Q_N = Q_N^1 \cup Q_N^2 \cup \{q_0\}$$
$$Q_E = Q_E^1 \cup Q_E^2 \cup \{q_f\},$$
$$\Delta = \Delta^1 \cup \Delta^2 \cup \{(q_0, \varphi, q) \mid (q_0^1, \varphi, q) \in \Delta^1 \text{ 或 } (q_0^2, \varphi, q) \in \Delta^2\}$$
$$\cup \{(q, \varphi, q_f) \mid (q, \varphi, q_f^1) \in \Delta^1 \text{ or } (q, \varphi, q_f^2) \in \Delta^2\}$$
$$\cup \{(q_0, \varphi, q_f) \mid (q_0^1, \varphi, q_f^1) \in \Delta^1 \text{ or } (q_0^2, \varphi, q_f^2) \in \Delta^2\}$$

- P $= \mathrm{P}_1^*$。状态集为 $Q_N = Q_N^1 \cup \{q_0\}$ 和 $Q_E = Q_E^1 \cup \{q_f\}$，其中 $q_0$ 和 $q_f$ 分别是新的初始状态和接受状态。转移关系为 $\Delta = \Delta^1 \cup \Delta_\varepsilon \cup \Delta_{\geq 1} \cup \Delta_t$，其中 $\Delta_\varepsilon$、$\Delta_{\geq 1}$ 和 $\Delta_t$ 的定义如下。

P 匹配一条空路径的情形，即零次重复的情形，由 $\Delta_\varepsilon$ 涵盖。回想一下，长度为零的路径由一个节点（带有任意标签）组成。因此，我们令 $\Delta_\varepsilon = \{(q_0, \top, q_f)\}$。

对于其他情况，将 $q_0^1$ 和 $q_f^1$ 的转移复制到 $q_0$ 和 $q_f$。

$$\Delta_{\geq 1} = \{(q_0, \varphi, q^1) \mid (q_0^1, \varphi, q^1) \in \Delta^1\}$$
$$\cup \{(q^1, \varphi, q_f) \mid (q^1, \varphi, q_f^1) \in \Delta^1\}$$

最后，我们允许自动机“循环”。与连接的构造类似，必须合并转移。

$$\Delta_t = \{(q_s^1, \varphi \wedge \psi, q_t^1) \mid (q_s^1, \varphi, q_f^1) \in \Delta^1, (q_0^1, \psi, q_t^1) \in \Delta^1\}$$

添加反向遍历。为处理反向遍历，即形如 $:\xleftarrow{\psi}$ 的模式，我们将把边的方向理解为一个附加标签。更确切地说，我们考虑两个特殊标签 fw 和 bw，分别用于正向遍历和反向遍历，它们不会出现在模式 P（或底层属性图）中。对于模式 $:\xrightarrow{\varphi}$，我们随后构造转移 $(q_s, \mathrm{fw} \wedge \varphi, q_t)$，而不是 $(q_s, \varphi, q_t)$。模式 $:\xleftarrow{\psi}$ 的自动机随后可以类似地构造，从而得到转移 $(q_s, \mathrm{bw} \wedge \varphi, q_t)$。请注意，在整体构造中，边模式对应转移的标签表达式从不合并（只有节点标签的标签表达式才会合并）。因此，$Q_E$ 中各状态的所有出转移，即表明下一次转移读取边的标签的那些状态，其标签表达式要么形如 $\mathrm{fw} \wedge \varphi$，要么形如 $\mathrm{bw} \wedge \varphi$，其中 $\varphi$ 和 $\psi$ 既不包含 fw 也不包含 bw。$Q_N$ 中各状态的出转移完全不包含特殊标签 fw 和 bw，因为它们只读取节点标签。

为调整语义，我们将路径 $P = v_0 e_1 v_1 \ldots e_n v_n$ 的扩展轨迹 定义为

$$\lambda^*(P) = \lambda(v_0)\lambda^*(v_0, e_1, v_1)\lambda(v_1) \ldots \lambda^*(v_{n-1}, e_n, v_n)\lambda(v_n),$$

其中，$\lambda(v_i)$ 与之前一样，是节点 $v_i$ 的标签集合，而

$$\lambda^*(v_{i-1}, e_i, v_i) = \begin{cases} \lambda(e_i) \cup \{\mathrm{fw}\}, & \text{if } \rho(e_i) = (v_{i-1}, v_i), \\ \lambda(e_i) \cup \{\mathrm{bw}\}, & \text{if } \rho(e_i) = (v_i, v_{i-1}) \end{cases}$$

则根据边 $e_i$ 在路径 $P$ 中是以正向还是反向方式遍历，将边的标签集合 $\lambda(e_i)$ 扩展为 fw 或 bw。

若 RGPC 自动机（允许反向遍历）接受路径 $P = v_0 e_1 v_1 \ldots e_n v_n$ 的扩展轨迹 $\lambda^*(P)$，则该自动机接受此路径。

## B.2 错误修复

本节专门证明命题 4.7和 4.8。

命题 4.7 的证明。设 $\Sigma$ 为一组 RGPC 约束，$G$ 为一个属性图，且 $V$ 为 $G$ 相对于 $\Sigma$ 的（拓扑）冲突超图的最小权顶点覆盖。此外，设 $G'$ 为从 $G$ 中移除 $V$ 中的所有节点和边，以及与 $V$ 中的节点相连的所有边后得到的子图。我们通过反证法证明 $G'$ 是一个修复，即证明它满足定义 4.3 中的条件 a) 和 b)。

为导出矛盾，假设条件 a) 不成立，也就是说，$G'$ 违反了来自 $\Sigma$ 的约束 $q; F \Rightarrow C$。那么，$G'$ 中存在一个与 $q$ 匹配的 $\mu$。设 $O_\mu$ 为由 $\mu$ 诱导的拓扑错误。由于 $G'$ 的所有节点和边也都以相同的标签和属性出现在 $G'$ 中——即 $G'$ 是一个拓扑子图——因此 $O_\mu$ 也是 $G$ 的一个拓扑错误。但它同时也是 $G$ 的冲突超图中的一条超边，并且由于它的所有元素仍然存在于 $G'$ 中，它与 $V$ 的交集为空。这与 $V$ 是一个顶点覆盖相矛盾。由此得出，条件 a) 成立。

为了证明条件 b) 成立，我们再次反证，假设存在 $G$ 的一个子图 $G''$，使得 $G'$

是 $G''$ 的真子图，且 $G''$ 满足来自 $\Sigma$ 的所有约束。于是， $G''$ 中存在一个节点或边 $o$，它不存在于 $G'$ 中。我们进行分类讨论。

对于情形 $o \in V$，考虑包含对象 $o$ 的所有拓扑错误 $O_1, \ldots, O_m$，它们属于 $G$。由于 $G''$ 满足所有约束，因此存在对象 $o_1 \in O_1, \ldots, o_m \in O_m$，它们与 $o$ 不同，且不在 $G''$ 中。但由于 $G'$ 是 $G''$ 的子图，对象 $o_1, \ldots, o_m$ 也不在 $G'$ 中。此外，我们可以假设每个 $o_j$ 都在 $V$ 中：否则，$o_j$ 是一条被删除的边，因为它的一个端点（比如节点 $n$）被删除了。但由于错误源自路径， $O_j$ 也包含 $n$。此外， $n$ 不可能是 $o$，因为 $n$ 的权重高于其关联边的权重之和，因此也高于所有不在 $V$ 中的 $o_j$ 的权重之和。因此， $n$ 在 $V$ 中，但某些边 $o_j$ 不在 $V$ 中，这与 $V$ 具有最小权重相矛盾。因此，我们确实可以假设 $o_j$ 是 $n$，因而属于 $V$。此外，由于所有 $o_j$ 都在 $V$ 中，我们可以得出结论：$V \setminus \{o\}$ 是一个顶点覆盖。由于它的权重小于 $V$，这与 $V$ 是极小顶点覆盖这一事实相矛盾。

接下来考虑情形 $o \notin V$。那么， $o$ 是一条边，它被移除是因为其端点之一 $n$ 因属于 $V$ 而被移除。由于 $G''$ 包含 $o$，它也必须包含其端点 $n$。因此，对 $n \in V$ 应用第一种情形会导出矛盾。

总之，我们可以得出结论：条件 b) 成立 . □

命题 4.8 的证明。我们采用与命题 4.7 证明中类似的方法。设 $\Sigma$ 为一组 RGPC 约束， $G$ 为一个属性图，且 $V_g$ 为其关于 $G$ 的（拓扑）冲突超图的一个最小顶点覆盖， $\Sigma$ 由贪心算法计算得到。此外，设 $G'$ 为从 $G$ 中移除 $V_g$ 的所有节点和边，以及与节点 $V_g$ 关联的所有边后得到的子图。我们通过证明 $G'$ 是一个修复，来说明定义 4.3 中的条件 a) 和 b) 均成立。条件 a) 成立的证明与命题 4.7 的证明完全相同，因为它只依赖于 $V_g$ 是一个顶点覆盖。

为证明条件 b) 成立，我们反设存在图 $G$ 的一个子图 $G''$，使得 $G'$ 是 $G''$ 的真子图，且 $G''$ 满足 $\Sigma$ 中的所有约束。那么， $G''$ 中存在一个节点或边 $o$，但它不在 $G'$ 中。

在情况 $o \in V_g$ 下，考虑 $G$ 中带有 $V_g \cap O_i = \{o\}$ 的所有拓扑错误 $O_1$、……、$O_m$。注意，至少存在一个这样的 $O_i$，否则 $o$ 就会在贪心算法的第二阶段中从 $V_g$ 中被移除。此外，至少对于这些错误中的一个 $O_j$， $o$ 是 $O_j$ 中所有对象里权重最小的对象。如果情况不是这样，贪心算法就会在第一阶段选取权重小于 $o$ 的元素 $o_1 \in O_1$、……、$o_m \in O_m$，从而在第二阶段中先于移除 $o_1$、……、$o_m$ 而从 $V_g$ 中移除 $o$，因为 $o$ 的权重更大。我们固定其中一个错误 $O_j$，使得 $o$ 在 $O_j$ 中的对象里具有最小权重。由于 $V_g \cap O_j = \{o\}$，所有对象 $O_j \setminus \{o\}$ 都属于 $G'$，因此也属于 $G''$，因为 $G'$ 是 $G''$ 的子图。注意，当 $o$ 是一个节点时，这一点尤其成立：由于 $o$ 具有最小权重， $O_j$ 在这种情况下不可能包含一条边。但是，由于 $G''$ 也包含 $o$，它包含 $O_j$ 的所有对象。这是一个矛盾，因为 $O_j$ 源自模式 $q$ 关于约束 $q$ 的匹配 $\mu$；$F \Rightarrow C$ 在 $\Sigma$ 中通过了 $F$，但不满足 $C$。因此， $G''$ 违反了该约束。

还需考虑情况 $o \notin V_g$。此时， $o$ 是一条边，它之所以被移除，是因为其一个端点 $n$ 因属于 $V_g$ 而被移除。由于 $G''$ 包含 $o$，它也必须包含其端点 $n$。因此，对 $n \in V_g$ 应用第一种情况可得矛盾。总之，我们可以得出结论：条件 b) 成立。□

## B.3 含显式依赖关系的整数线性规划

本节讨论一种不同于第 4.1 节所述整数线性规划的方案，以及我们为何没有选择该方案。得益于第 4.1 节所述整数线性规划中权重的使用，可以确保只有在必要时才删除节点，并且优先删除（其子集中的）关联边。不过，这种方法的一个潜在问题是权重固定不变（即在整数线性规划中硬编码）。例如，考虑一条约束，它禁止两个不同的节点具有相同的名称属性值。假设有两个节点 $n_1$、$n_2$，它们分别有两条和五条关联边，违反了这条约束。由于 $n_1$ 的权重为 3，而 $n_2$ 的权重为 6，整数线性规划求解器总会将 $n_1$ 标记为待删除。尤其是，即便由于另一条约束，整数线性规划也将 $n_2$ 的所有关联边标记为待删除（而没有将 $n_1$ 的任何关联边标记为待删除），情况依然如此。但在这种情况下， $n_2$ 的“有效权重”仅为 1，因此应当将 $n_2$ 而非 $n_1$ 标记为待删除。对此，可以构造并求解另一种整数线性规划，其中显式编码依赖关系。也就是说，如果属性图中的 $n_1$ 有两条关联边 $e_1$ 和 $e_2$，那么我们就在整数线性规划中加入约束 $2x_{n_1} \leq x_{e_1} + x_{e_2}$。有了这条约束，只要表示 $n_1$ 的变量 $x_{n_1}$ 在解中被赋值为 1，表示这些边的变量 $x_{e_1}$ 和 $x_{e_2}$ 就必须被赋值为 1（请记住，值 1 表示对象被删除）。这样一来便不再需要权重，整数线性规划的形式如下，其中 $E_n$ 是属性图中节点 $n \in N$ 的关联边集合。[9]

$$
\begin{aligned}
\text{minimize} \quad & \sum_{v \in \mathcal{W}} x_v \\
\text{subject to} \quad & \sum_{v \in O} x_v \geq 1 \quad \text{for all } O \in \mathcal{E} \\
\text{and} \quad & |E_n| \cdot x_n \leq \sum_{e \in E_n} x_e \quad \text{for all } n \in N \cap \mathcal{W} \\
\text{and} \quad & 0 \leq x_v \leq 1 \quad \text{for all } v \in \mathcal{W}
\end{aligned}
$$

该变体以更大的整数线性规划换取潜在更好的修复。然而，在我们的实验中，在所有情况下求解带显式依赖的整数线性规划都显著更耗时，而得到的修复与带权重的整数线性规划相同。

## C 实验附加材料

在附录C.1 中，我们展示了第5节未展示的可选步骤／数据集／约束组合的图表。附录C.2包含绘图所用的原始数据，以及运行时间增加和删除减少百分比，尤其是论文中报告的那些数据。

[9]严格来说，每个 $E_n$ 与 $n \in \mathcal{W}$ 的元素都必须添加到冲突超图的顶点集 $\mathcal{W}$ 中。

## C.1 补充实验结果

我们为其余可选步骤／数据集／约束组合提供了更多图表，形式与第5节中的图表类似。

扩展性。 在第 5.1 节中，我们提供了图表，展示基础流水线在 ICIJ 和 LDBC 图上处理 1 路约束时的扩展性。图 13 展示了立法和 Coreutils 数据集的类似图表。正如论文中对 ICIJ 和 LDBC 数据集的观察，删除数量相对较少，且流水线具有良好的扩展性。

我们还针对启用步骤 1 路约束的步骤 2 进行了扩展性实验。图 17 和18 展示了启用步骤 2 时的运行时间和标签删除数量。结果表明，与基础流水线一样，启用步骤 2 时，如果所需删除数量较少，我们的流水线也能表现良好；不过其运行时间始终高于基础流水线，这印证了我们在第 5.3 节中的观察。

近似修复。 在第 5.1 节中，我们研究了朴素贪心算法和线性规划引导的贪心算法的选择阶段所得到的近似修复。对于 ICIJ 图，我们指出，朴素贪心算法的选择阶段通常优于线性规划引导的贪心算法的选择阶段，但总体而言，朴素贪心算法表现较差，因为它在修剪阶段必须修剪更多对象。图 15 和 16展示了其他数据集上的相同情况；与论文中的图6a、6b、6c 和 6d（展示总运行时间）以及图 7a、7b、7c 和 7d（展示完整算法计算修复时的边删除数量）相比，也可得出这一结论。与论文中对 ICIJ 数据集的观察一样，我们还可以发现，线性规划引导的贪心算法往往能达到最优质量（因此实际计算出一个修复），而朴素贪心算法则不能。然而，当朴素贪心算法在选择阶段得到的结果接近最优解时，它在计算修复（而不仅仅是近似修复）方面可能优于线性规划引导的贪心算法。

最小化重要对象的删除。在第5.2 节中，我们评估了流水线对自定义权重的鲁棒性。具体而言，我们使用节点和边的 PageRank 分数作为自定义权重来修复 ICIJ 图，并指出使用自定义权重时，修复的权重损失可减少24% 。这表明，更重要的边（即 PageRank 分数较高的边）得以保留。表3展示了所有约束集的总权重、权重损失和损失减少量。损失减少量的计算方式如下：

$$100 \cdot \left(1 - \frac{w_{\text{orig}} - w_{\text{cw-repair}}}{w_{\text{orig}} - w_{\text{repair}}}\right)$$

其中， $w_{\text{orig}}$ 表示输入图的总权重， $w_{\text{cw-repair}}$ 表示提供自定义权重时修复结果的总权重，$w_{\text{repair}}$ 表示不提供自定义权重时修复结果的总权重。对于 2-rep 约束，损失减少量甚至达到 46 %；不过，与论文中讨论的单向约束情况相比，不使用自定义权重时修复结果的权重损失更低。

表 3还展示了 Coreutils图的实验结果，图 14a 和 14b 展示了在提供 PageRank 分数作为自定义权重时的运行时间和边删除情况。与论文中的图 6b 和 7b 相比，它们证实了我们对 ICIJ 图的发现。

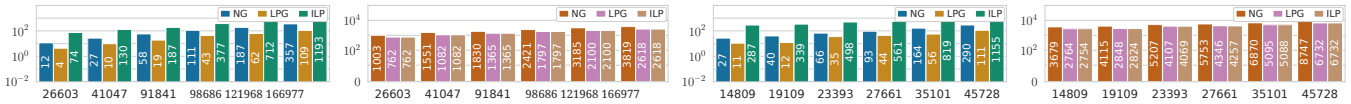
我们使用 Neo4j 的图数据科学库来计算 PageRank 分数。由于通常是为节点计算 PageRank 分数，我们使用线图来计算边的 PageRank 分数。在这种图中，节点和边的作用直观上是互换的：图 $G = (V, E)$ 的线图 $G_L = (V_L, E_L)$是一个包含$V_L = E$的图，并且当且仅当存在$v_0, v_1, v_2 \in V$使得 $v_0e_1v_1e_2v_2$ 是 $G$中的一条路径时，$(e_1, e_2) \in V_L \times V_L$ 属于 $E_L$。对于立法图和 LDBC 图，由于硬件条件限制，我们无法使用 Neo4j 构建线图来计算边权重。

步骤 *2*／标签删除。 图 19 展示了在 Coreutils 和 LDBC 数据集以及表 1中列出的约束集上，启用步骤 2 后的修复流水线运行时间和边删除数量。与第 5.4 节中基础流水线对应的图相比，这些结果与第 5.3 节讨论的立法图和 ICIJ 图的类似结果揭示了相同的情况：启用步骤 2 后，修复流水线能获得质量更高的修复（以删除数量衡量），但运行时间也会更长。

步骤 *3*／邻域错误。图 20、21、21 和23展示了邻域错误修复（步骤 3）的绘图，但 ICIJ 图与线性规划引导的贪心算法的组合除外；该组合的结果见第 5.3 节。这些图支持我们在第 5.3 节讨论的见解：如果 $k$ 选择得当，几乎在所有情况下都能在保持高质量的同时缩短运行时间。对于 2-rep 约束（仅适用于 ICIJ 图和 LDBC 图）， $k = 4$ 的结果质量接近（甚至与）基础流水线相同；对于其他约束， $k = 2$ 就足够了。这种效应可以用约束的形状来解释：2-rep 约束的匹配中，路径的最小长度为 9，而其他约束中的路径长度至多为 6。对于所有路径长度都不超过 6 的匹配，完整匹配都处于 4-邻域内，而 2-邻域覆盖了其中的大部分。此外，由于我们的约束具有重复，因此较大的错误很可能通过修复较小的错误来解决，尤其是那些包含长度至多为 6 的路径的错误。对于 2-rep 约束，4-邻域几乎完整覆盖了包含最短路径的匹配，但 2-邻域无法做到这一点；这解释了为什么对于 $k = 4$，结果与基准情况非常接近，而在这种情况下， $k = 2$ 的结果较差。这表明， $k$的一个良好取值是 $k = \ell_{\text{最小}2}$，其中 $\ell_{\min}$ 是约束的路径模式所匹配路径的最小长度。

另一个值得注意的观察是，与未启用步骤 3相比，启用步骤 3时，朴素贪心算法有时能获得更高的质量。例如，对于 1元约束以及 $k = 4$ 立法图（论文中的图 20d与图 7a ）会出现这种情况。其根本原因在于，朴素贪心算法会从错误中任意选择一个对象进行移除：因此，截断错误可能会改变所选对象，并带来质量更高的修复（但在其他情况下也可能导致质量更低的修复）。对于线性规划引导的贪心算法和整数线性规划策略，我们无法观察到这种影响，因为借助（整数）线性规划解，这些策略能够更好地避免错误选择。

最后，图 24 展示了使用随机抽取的 $2k$条边运行流水线时的结果，这些边来自每种拓扑错误，但




(a) 立法：运行时间　(b) 立法：边删除　(c) Coreutils：运行时间　(d) Coreutils：边删除

**图 13：运行时间和　LDBC 和 Coreutils图的边删除数量，针对1路约束子集　s**

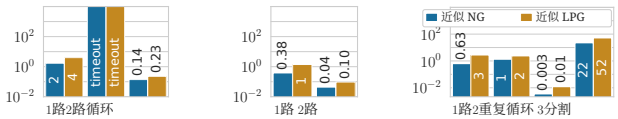
**表 3：流水线使用或不使用自定义权重时，线性规划引导的贪心算法提出的修复方案的边权重总和。此处边的权重为其在线图中的 PageRank。为便于阅读，所有权重均已四舍五入。**

| 数据集 | 形状 | 原始权重 | 不使用自定义权重的修复 | | 使用自定义权重的修复 | | |
|---|---|---|---|---|---|---|---|
| | | | 修复权重 | 权重损失 | 修复权重 | 权重损失 | 损失减少 |
| ICIJ | 单向 | 579593.37 | 579060.61 | 0.0919% | 579188.60 | 0.0698% | 24% |
| | 双向 | | 579233.32 | 0.0621% | 579261.99 | 0.0572% | 7% |
| | 两次重复 | | 579548.82 | 0.0077% | 579569.60 | 0.0041% | 46% |
| | loop | | 579143.61 | 0.0776% | 579219.41 | 0.0645% | 16% |
| | 三分 | | 579590.94 | 0.0004% | 579590.94 | 0.0004% | 0% |
| Coreutils | 单向 | 112369.10 | 110873.74 | 1.3308% | 110920.13 | 1.2895% | 3% |
| | 双向 | | 112203.69 | 0.1472% | 112225.11 | 0.1281% | 13% |



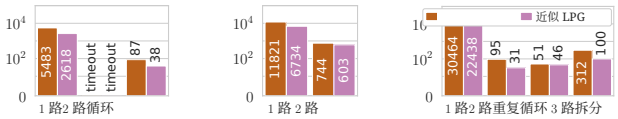

(a) 运行时间　(b) 边删除

**图14：使用 PageRank 分数作为自定义权重时，Core-utils 图的运行时间（秒）和删除次数**




(a) 立法　(b) Coreutils　(c) LDBC

**图 15：朴素贪心算法 (NG) 和线性规划引导的贪心算法 (LPG) 选择阶段的运行时间（秒）**




(a) 立法　(b) Coreutils　(c) LDBC

**图 16：朴素贪心算法（NG）和线性规划引导的贪心算法（LPG）选择阶段的边删除数量**

ICIJ 图除外，其结果已在论文第 5.3 节中展示。这些结果证实了论文中讨论的观察：对于 ICIJ 图，随机抽样可能会导致更多删除。我们得出结论，围绕路径端点选取邻域是更合适的策略。

## C.2 原始测量值与权衡百分比

表 4 展示了表 1 中所列数据集和约束集的运行时间，分别对应基础流水线以及启用步骤 2（标签删除）的情况，结果见第四列和第六列。所有数值均为四次运行的平均值。第 5 列给出了相对于整数线性规划算法，朴素贪心算法和线性规划引导的贪心算法为基础流水线带来的运行时间缩减百分比。具体而言，运行时间缩减为

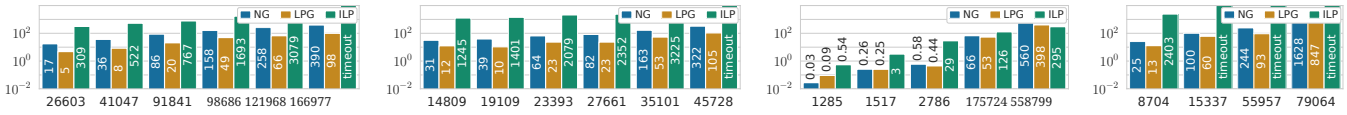
$$100 \cdot \frac{r_{\mathtt{ilp}} - r_{\mathtt{g}}}{r_{\mathtt{ilp}}},$$

其中，$r_{\mathtt{ilp}}$ 是整数线性规划算法的运行时间，$r_{\mathtt{g}}$ 是朴素贪心算法或线性规划引导的贪心算法的运行时间。例如，前三行表明，对于 Coreutils 图上的单向约束，朴素贪心算法的运行时间比整数线性规划算法的参考值 1154.86$s$ 低 74.86%，而线性规划引导的贪心算法甚至实现了 90.13% 的运行时间缩减。

同样，最后一列显示了运行时间的增加

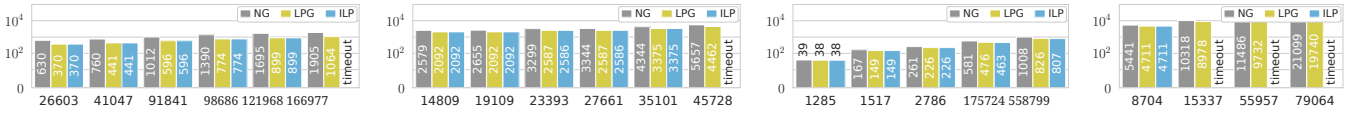
$$100 \cdot \frac{r_{\mathtt{label}}}{r_{\mathtt{base}}},$$

启用步骤 2 时的情况。在此处 $r_{\mathtt{base}}$ 表示基础流水线中朴素贪心算法、线性规划引导的贪心算法或整数线性规划算法的运行时间，而 $r_{\mathtt{label}}$ 表示启用步骤 2 后的运行时间。例如，第一行表示，对于基础流水线，朴素贪心算法耗时 290.32$s$，而启用步骤 2 后耗时 320.46$s$，即 110。38% 的参考值 290.32$s$。极值，尤其是正文中报告的数字，会以粗体显示在（蓝色）边框中。表 10 和11 展示了扩展性图中对应的数字，表 15 展示了用于计算近似修复的贪心算法选择阶段的对应数字，相关讨论见第 5.1 节和附录 C.1。对于第 5.2 节和附录 C.1中的自定义权重实验，结果列于表 14。




(a) 立法 (b) Coreutils (c) ICIJ (d) LDBC

图 17：针对1路约束子集启用 Step2（标签删除）后的修复流程运行时间



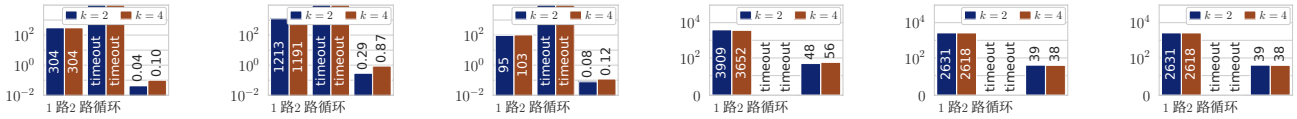

(a) 立法 (b) Coreutils (c) ICIJ (d) LDBC

图18：对部分 1 元约束启用步骤 2 （标签删除）后，修复流程提出的标签删除



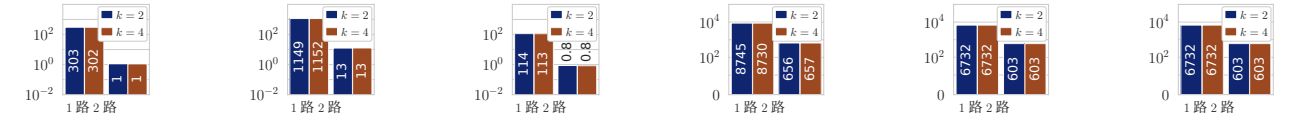

(a) Coreutils：运行时间 (b) LDBC：运行时间 (c) Coreutils：标签删除 (d) LDBC：标签删除

图19：启用步骤 2 时，Coreutils图和 LDBC 的运行时间及标签删除数量




(a) 天然气：运行时间 (b) 整数线性规划：运行时间 (c) 液化石油气：运行时间 (d) 天然气：边删除 (e) 整数线性规划：边删除 (f) 液化石油气：边删除

图20：立法图的邻域错误对应的运行时间（秒）和边删除数量



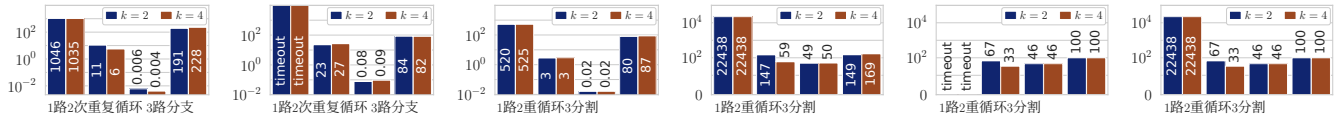

(a) 天然气：运行时间 (b) 整数线性规划：运行时间 (c) 液化石油气：运行时间 (d) 天然气：边删除 (e) 整数线性规划：边删除 (f) 液化石油气：边删除

图 21：Coreutils图的邻域错误对应的运行时间（秒）和边删除数



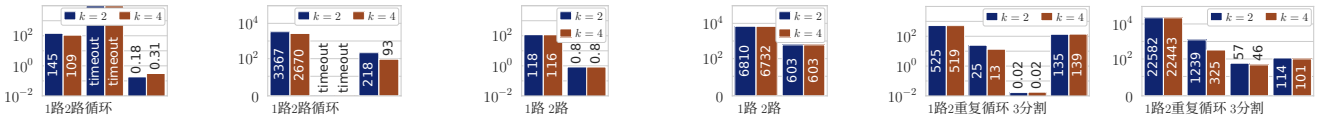

(a) 天然气：运行时间 (b) 整数线性规划：运行时间 (c) 天然气：边删除 (d) 整数线性规划：边删除

图 22：ICIJ 图中邻域错误的运行时间（秒）和边删除




(a) 天然气：运行时间 (b) 整数线性规划：运行时间 (c) 液化石油气：运行时间 (d) 天然气：边删除 (e) 整数线性规划：边删除 (f) 液化石油气：边删除

Figure 23：LDBC图邻域错误的运行时间（秒）和边删除




**(a) 立法：运行时间 (b) 立法：删除 (c) Coreutils：运行时间 (d) Coreutils：删除** **(e) LDBC：运行时间** **(f) LDBC：删除**

**图 24：线性规划引导的贪心算法针对样例错误的运行时间（秒）和边删除数量**

表5 展示了基础流水线以及启用步骤 2 时记录的删除数量。例如，表 5 的前三行显示，朴素贪心算法产生8747 条边删除，而线性规划引导的贪心算法和整数线性规划算法在基础流水线中均产生 6732 条边删除。也就是说，线性规划引导的贪心算法和整数线性规划策略产生的删除量减少了 23。04%，这一数据列于第五列，因此质量更高。一般而言，删除量减少的计算方式如下：

$$100 \cdot \frac{d_{\mathsf{ng}} - d_{\mathsf{lp}}}{d_{\mathsf{ng}}},$$

其中， $d_{\mathsf{ng}}$ 是朴素贪心算法产生的边删除数量， $d_{\mathsf{lp}}$ 是线性规划引导的贪心算法或整数线性规划策略产生的删除数量。最后一列显示启用步骤 2 后删除数量的减少（或质量增益），计算方式如下：

$$100 \cdot \frac{d_{\mathsf{base}} - d_{\mathsf{label}}}{d_{\mathsf{base}}},$$

其中， $d_{\mathsf{base}}$ 是基础流水线产生的边删除数量， $d_{\mathsf{label}}$ 是在启用步骤 2 后产生的标签删除数量。例如，第一行说明，启用步骤 2 后，朴素贪心算法选择删除约 5655 个标签（第 6 列），比第 4 列中的 8747 次边删除这一参考值少 35.35%。至于运行时间表，所有数值均为四次运行的平均值。由于贪心算法每次运行选择的删除数量可能不同，因此平均值不一定是整数。表12和 13展示了扩展性实验中的删除数量，表 16展示了第 5.1 节和附录 C.1中讨论的近似修复实验结果。第 5.2 节和附录 C.1中使用自定义权重的实验结果见表 14。

表 6 展示了步骤 3（邻域错误修复）的运行时间及相对于基础流水线的运行时间缩减；计算方法与上文表 4 中说明的方法类似。运行时间缩减按以下公式计算

$$100 \cdot \frac{r_{\mathsf{base}} - r_{\mathsf{nbh}}}{r_{\mathsf{base}}},$$

其中 $r_{\mathsf{nbh}}$ 和 $r_{\mathsf{base}}$ 分别表示启用和未启用步骤 3 时的运行时间。第 5.3 节中使用采样错误的实验的运行时间（缩减）见表 8。

最后，表 7 展示了启用和未启用步骤 3 时的删除数量。百分比按以下公式计算

$$100 \cdot \frac{d_{\mathsf{nbh}}}{d_{\mathsf{base}}},$$

其中 $d_{\mathsf{nbh}}$ 和 $d_{\mathsf{base}}$ 分别表示启用步骤 3 和未启用步骤 3 时的删除数量。第 5.3 节中使用采样错误的实验的边删除数量见表 9。

**表 4：表中所列数据集和约束集的原始运行时间及运行时间比较 1**

| | | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | 运行时间 | 运行时间缩减 | 运行时间 | 运行时间增加 |
| Coreutils | 单向 | 45728 | 朴素贪心算法 | 290.32 | 74.86 | 320.46 | 110.38 |
| | 单向 | 45728 | 线性规划引导的贪心算法 | 113.95 | 90.13 | 109.13 | 95.78 |
| | 单向 | 45728 | ILP | 1154.86 | | | |
| | 双向 | 3707 | 朴素贪心算法 | 1.11 | 91.19 | 2.11 | 190.02 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 0.85 | 93.24 | 1.56 | 182.73 |
| | 双向 | 3707 | ILP | 12.59 | | 49.92 | 396.47 |
| 立法 | 单向 | 166977 | 朴素贪心算法 | 356.64 | 70.10 | 389.81 | 109.30 |
| | 单向 | 166977 | 线性规划引导的贪心算法 | 111.94 | 90.62 | 97.90 | 87.46 |
| | 单向 | 166977 | ILP | 1192.84 | | | |
| | 双向 | 419439 | 朴素贪心算法 | | | | |
| | 双向 | 443695 | 线性规划引导的贪心算法 | | | | |
| | 双向 | 419439 | ILP | | | | |
| | loop | 3783 | 朴素贪心算法 | 0.18 | 86.57 | 0.22 | 126.09 |
| | loop | 3783 | 线性规划引导的贪心算法 | 0.28 | 78.61 | 0.56 | 200.18 |
| | loop | 3783 | ILP | 1.31 | | 4.37 | 334.05 |
| ICIJ | 单向 | 558799 | 朴素贪心算法 | 324.12 | **-445.65** | 559.79 | 172.71 |
| | 单向 | 558799 | 线性规划引导的贪心算法 | 183.48 | -208.89 | 393.31 | **214.36** |
| | 单向 | 558799 | ILP | 59.40 | | 294.69 | **496.11** |
| | 两次重复 | 397108 | 朴素贪心算法 | 35.68 | | 53.62 | 150.28 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 45.27 | | | |
| | 两次重复 | 397108 | ILP | | | | |
| | 双向 | 7553 | 朴素贪心算法 | 4.89 | **95.51** | 6.52 | 133.22 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 2.34 | **97.85** | 3.75 | 159.83 |
| | 双向 | 7553 | ILP | 108.90 | | 429.15 | 394.07 |
| | loop | 56614 | 朴素贪心算法 | 6.14 | 10.13 | 14.69 | 239.29 |
| | loop | 56614 | 线性规划引导的贪心算法 | 7.51 | -9.97 | 16.04 | 213.45 |
| | loop | 56614 | ILP | 6.83 | | 23.21 | 339.75 |
| | 三分 | 876933 | 朴素贪心算法 | 3.03 | 89.02 | 4.11 | 135.57 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 25.98 | 5.91 | 54.69 | 210.54 |
| | 三分 | 876933 | ILP | 27.61 | | 56.13 | 203.29 |
| LDBC | 单向 | 79064 | 朴素贪心算法 | 980.82 | | 1627.58 | 165.94 |
| | 单向 | 79064 | 线性规划引导的贪心算法 | 527.74 | | 846.50 | 160.40 |
| | 单向 | 79064 | ILP | | | | |
| | 两次重复 | 72756 | 朴素贪心算法 | 3.90 | 86.29 | 9.99 | **256.01** |
| | 两次重复 | 72756 | 线性规划引导的贪心算法 | 3.36 | 88.18 | 6.35 | 188.79 |
| | 两次重复 | 72756 | ILP | 28.46 | | 109.26 | 383.86 |
| | loop | 423 | 朴素贪心算法 | 0.0041 | **95.97** | 0.01 | 182.72 |
| | loop | 423 | 线性规划引导的贪心算法 | 0.01 | 86.28 | 0.02 | 162.33 |
| | loop | 423 | ILP | 0.10 | | 0.35 | 347.20 |
| | 三分 | 1358499 | 朴素贪心算法 | 225.46 | -165.99 | 477.74 | 211.90 |
| | 三分 | 1358499 | 线性规划引导的贪心算法 | 87.64 | -3.40 | 173.72 | 198.22 |
| | 三分 | 1358499 | ILP | 84.76 | | 143.25 | 169.00 |

**表 5：表中所列数据集和约束集的删除次数及比较次数1**

| | | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | #边删除 | 删除量减少 | #标签删除 | 删除减少 |
| Coreutils | 单向 | 45728 | 朴素贪心算法 | 8747.00 | | 5655.29 | 35.35 |
| | 单向 | 45728 | 线性规划引导的贪心算法 | 6732.00 | 23.04 | 4462.00 | 33.72 |
| | 单向 | 45728 | ILP | 6732.00 | 23.04 | | |
| | 双向 | 3707 | 朴素贪心算法 | 656.00 | | 626.33 | 4.52 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 603.00 | 8.08 | 593.00 | 1.66 |
| | 双向 | 3707 | ILP | 603.00 | 8.08 | 593.00 | 1.66 |
| Legislative | 单向 | 166977 | 朴素贪心算法 | 3818.67 | | 1905.33 | **50.10** |
| | 单向 | 166977 | 线性规划引导的贪心算法 | 2618.00 | 31.44 | 1064.33 | **59.35** |
| | 单向 | 166977 | ILP | 2618.00 | 31.44 | | |
| | 双向 | 419439 | 朴素贪心算法 | | | | |
| | 双向 | 443695 | 线性规划引导的贪心算法 | | | | |
| | 双向 | 419439 | ILP | | | | |
| | loop | 3783 | 朴素贪心算法 | 49.67 | | 32.33 | 34.90 |
| | loop | 3783 | 线性规划引导的贪心算法 | 38.00 | 23.49 | 20.00 | 47.37 |
| | loop | 3783 | ILP | 38.00 | 23.49 | 20.00 | **47.37** |
| ICIJ | 单向 | 558799 | 朴素贪心算法 | 1411.00 | | 1008.33 | 28.54 |
| | 单向 | 558799 | 线性规划引导的贪心算法 | 1188.33 | 15.78 | 826.33 | 30.46 |
| | 单向 | 558799 | ILP | 1188.00 | 15.80 | 807.00 | 32.07 |
| | 两次重复 | 397108 | 朴素贪心算法 | 79.67 | | 57.00 | 28.45 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 50.67 | **36.40** | | |
| | 两次重复 | 397108 | ILP | | | | |
| | 双向 | 7553 | 朴素贪心算法 | 1588.33 | | 1261.00 | 20.61 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 1176.33 | 25.94 | 1072.00 | 8.87 |
| | 双向 | 7553 | ILP | 1175.00 | 26.023 | 1072.00 | 8.77 |
| | loop | 56614 | 朴素贪心算法 | 727.33 | | 720.67 | 0.92 |
| | loop | 56614 | 线性规划引导的贪心算法 | 725.00 | 0.32 | 715.33 | 1.33 |
| | loop | 56614 | ILP | 725.00 | 0.32 | 715.00 | 1.38 |
| | 三分 | 876933 | 朴素贪心算法 | 2.00 | | 1.00 | 50.00 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 1.00 | 50.0 | 1.00 | 0.00 |
| | 三分 | 876933 | ILP | 1.00 | **50.0** | 1.00 | 0.00 |
| LDBC | 单向 | 79064 | 朴素贪心算法 | 22438.00 | | 21099.00 | 5.97 |
| | 单向 | 79064 | 线性规划引导的贪心算法 | 22438.00 | | 19740.00 | 12.02 |
| | 单向 | 79064 | ILP | | | | |
| | 两次重复 | 72756 | 朴素贪心算法 | 42.33 | | 51.33 | -21.26 |
| | 两次重复 | 72756 | 线性规划引导的贪心算法 | 31.00 | 26.77 | 30.00 | 3.23 |
| | 两次重复 | 72756 | ILP | 31.00 | 26.77 | 30.00 | 3.23 |
| | loop | 423 | 朴素贪心算法 | 48.67 | | 48.00 | 1.37 |
| | loop | 423 | 线性规划引导的贪心算法 | 46.00 | 5.48 | 44.00 | 4.35 |
| | loop | 423 | ILP | 46.00 | 5.48 | 44.00 | 4.35 |
| | 三分 | 1358499 | NG | 155.00 | | 142.33 | 8.17 |
| | 三分 | 1358499 | LPG | 100.00 | **35.48** | 93.00 | 7.00 |
| | 三分 | 1358499 | ILP | 100.00 | **35.48** | 93.00 | 7.00 |

**表 6：步骤 3（邻域错误）的原始运行时间和运行时间比较**

| | | | | 基础流水线 | | $k = 2$ | | $k = 4$ |
|---|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | 运行时间 | 运行时间 | 运行时间缩减 | 运行时间 | 运行时间缩减 |
| Coreutils | 单向 | 45728 | 朴素贪心算法 | 290.32 | 302.98 | -4.36 | 301.72 | -3.93 |
| | 单向 | 45728 | 线性规划引导的贪心算法 | 113.95 | 114.42 | -0.42 | 113.00 | 0.83 |
| | 单向 | 45728 | ILP | 1154.86 | 1148.77 | 0.53 | 1152.01 | 0.25 |
| | 双向 | 3707 | 朴素贪心算法 | 1.11 | 1.13 | -1.77 | 1.12 | -0.61 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 0.85 | 0.87 | -1.6445 | 0.84 | 1.01 |
| | 双向 | 3707 | ILP | 12.59 | 12.54 | 0.41 | 12.76 | -1.32 |
| Legislative | 单向 | 166977 | 朴素贪心算法 | 356.64 | 304.48 | 14.63 | 303.79 | 14.82 |
| | 单向 | 166977 | 线性规划引导的贪心算法 | 111.94 | 95.14 | 15.00 | 102.70 | 8.25 |
| | 单向 | 166977 | ILP | 1192.84 | 1213.44 | -1.73 | 1190.86 | 0.17 |
| | 双向 | 443695 | 朴素贪心算法 | | | | | |
| | 双向 | 443695 | 线性规划引导的贪心算法 | | | | | |
| | 双向 | 443695 | ILP | | | | | |
| | loop | 3783 | 朴素贪心算法 | 0.18 | 0.04 | 75.99 | 0.10 | 43.61 |
| | loop | 3783 | 线性规划引导的贪心算法 | 0.28 | 0.08 | 71.90 | 0.12 | 57.24 |
| | loop | 3783 | ILP | 1.31 | 0.29 | **78.04** | 0.87 | 33.86 |
| ICIJ | 单向 | 558799 | 朴素贪心算法 | 324.12 | 126.14 | 61.08 | 194.55 | 39.98 |
| | 单向 | 558799 | 线性规划引导的贪心算法 | 183.48 | 68.21 | **62.83** | 103.09 | 43.81 |
| | 单向 | 558799 | ILP | 59.40 | 41.21 | 30.62 | 46.21 | 22.21 |
| | 两次重复 | 397108 | 朴素贪心算法 | 35.68 | 26.33 | 26.20 | 30.24 | 15.26 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 45.27 | 9.19 | 79.69 | 44.98 | 0.63 |
| | 两次重复 | 397108 | ILP | | 8.78 | | | |
| | 双向 | 7553 | 朴素贪心算法 | 4.89 | 4.33 | 11.53 | 5.07 | -3.60 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 2.34 | 2.27 | 3.35 | 2.47 | -5.12 |
| | 双向 | 7553 | ILP | 108.90 | 106.91 | 1.83 | 107.20 | 1.57 |
| | loop | 56614 | 朴素贪心算法 | 6.14 | 2.90 | 52.84 | 4.23 | 31.08 |
| | loop | 56614 | 线性规划引导的贪心算法 | 7.51 | 3.50 | 53.42 | 5.01 | 33.36 |
| | loop | 56614 | ILP | 6.83 | 5.66 | 17.21 | 5.87 | 14.03 |
| | 三分 | 876933 | 朴素贪心算法 | 3.03 | 4.46 | -47.12 | 2.01 | 33.65 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 25.98 | 23.84 | 8.24 | 25.79 | 0.74 |
| | 三分 | 876933 | ILP | 27.61 | 25.35 | 8.19 | 26.83 | 2.84 |
| LDBC | 单向 | 79064 | 朴素贪心算法 | 980.82 | 1045.71 | -6.62 | 1034.62 | -5.48 |
| | 单向 | 79064 | 线性规划引导的贪心算法 | 527.74 | 519.94 | 1.48 | 524.52 | 0.61 |
| | 单向 | 79064 | ILP | | | | | |
| | 两次重复 | 72756 | 朴素贪心算法 | 3.90 | 10.76 | -175.78 | 5.58 | -42.98 |
| | 两次重复 | 72756 | ILP | 28.46 | 22.60 | 20.61 | 26.62 | 6.47 |
| | loop | 423 | 朴素贪心算法 | 0.0041 | 0.006 | -46.73 | 0.0041 | 1.02 |
| | loop | 423 | 线性规划引导的贪心算法 | 0.01 | 0.06 | -13.40 | 0.02 | -14.46 |
| | loop | 423 | ILP | 0.10 | 0.08 | 24.36 | 0.09 | 8.13 |
| | 三分 | 1358499 | 朴素贪心算法 | 225.46 | 191.02 | 15.2738 | 227.6318 | -0.9631 |
| | 三分 | 1358499 | 线性规划引导的贪心算法 | 87.64 | 79.87 | 8.86 | 87.07 | 0.65 |
| | 三分 | 1358499 | ILP | 84.76 | 83.97 | 0.93 | 82.45 | 2.73 |

**表 7：步骤 3（邻域错误）的原始边删除和删除比较**

| | | | | 基础流水线 | k = 2 | | k = 4 | |
|---|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | 删除 | 删除 | 百分比 | 删除 | 百分比 |
| Coreutils | 单向 | 45728 | 朴素贪心算法 | 8747.00 | 8744.67 | 99.97 | 8730.00 | 99.81 |
| | 单向 | 45728 | 线性规划引导的贪心算法 | 6732.00 | 6732.00 | 100.00 | 6732.00 | 100.00 |
| | 单向 | 45728 | ILP | 6732.00 | 6732.00 | 100.00 | 6732.00 | 100.00 |
| | 双向 | 3707 | 朴素贪心算法 | 656.00 | 656.00 | 100.00 | 656.67 | 100.10 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 603.00 | 603.00 | 100.00 | 603.00 | 100.00 |
| | 双向 | 3707 | ILP | 603.00 | 603.00 | 100.00 | 603.00 | 100.00 |
| 立法 | 单向 | 166977 | 朴素贪心算法 | 3818.67 | 3909.33 | 102.37 | 3652.00 | 95.64 |
| | 单向 | 166977 | 线性规划引导的贪心算法 | 2618.00 | 2631.00 | 100.50 | 2618.00 | 100.00 |
| | 单向 | 166977 | ILP | 2618.00 | 2631.00 | 100.50 | 2618.00 | 100.00 |
| | 双向 | 443695 | 朴素贪心算法 | | | | | |
| | 双向 | 443695 | 线性规划引导的贪心算法 | | | | | |
| | 双向 | 443695 | ILP | | | | | |
| | loop | 3783 | 朴素贪心算法 | 49.67 | 48.33 | 97.32 | 56.00 | 112.75 |
| | loop | 3783 | 线性规划引导的贪心算法 | 38.00 | 39.00 | 102.63 | 38.00 | 100.00 |
| | loop | 3783 | ILP | 38.00 | 39.00 | 102.63 | 38.00 | 100.00 |
| ICIJ | 单向 | 558799 | 朴素贪心算法 | 1411.00 | 1381.33 | 97.90 | 1438.33 | 101.94 |
| | 单向 | 558799 | 线性规划引导的贪心算法 | 1188.33 | 1188.33 | 100.00 | 1188.33 | 100.00 |
| | 单向 | 558799 | ILP | 1188.00 | 1188.00 | 100.00 | 1188.00 | 100.00 |
| | 两次重复 | 397108 | 朴素贪心算法 | 79.67 | 107.33 | 134.73 | 88.33 | 110.88 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 50.67 | 64.00 | 126.32 | 56.00 | 110.53 |
| | 两次重复 | 397108 | ILP | | 64.00 | | | |
| | 双向 | 7553 | 朴素贪心算法 | 1588.33 | 1552.67 | 97.75 | 1650.67 | 103.92 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 1176.33 | 1176.33 | 100.00 | 1175.33 | 99.91 |
| | 双向 | 7553 | ILP | 1175.00 | 1175.00 | 100.00 | 1175.00 | 100.00 |
| | loop | 56614 | 朴素贪心算法 | 727.33 | 727.67 | 100.05 | 730.33 | 100.41 |
| | loop | 56614 | 线性规划引导的贪心算法 | 725.00 | 725.00 | 100.00 | 725.00 | 100.00 |
| | loop | 56614 | ILP | 725.00 | 725.00 | 100.00 | 725.00 | 100.00 |
| | 三分 | 876933 | 朴素贪心算法 | 2.00 | 5.00 | 250.00 | 3.33 | 166.67 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 1.00 | 2.33 | 233.33 | 1.00 | 100.00 |
| | 三分 | 876933 | ILP | 1.00 | 2.00 | 200.00 | 1.00 | 100.00 |
| LDBC | 单向 | 79064 | 朴素贪心算法 | 22438.00 | 22438.00 | 100.00 | 22438.00 | 100.00 |
| | 单向 | 79064 | 线性规划引导的贪心算法 | 22438.00 | 22438.00 | 100.00 | 22438.00 | 100.00 |
| | 单向 | 79064 | ILP | | | | | |
| | 两次重复 | 72756 | 朴素贪心算法 | 42.33 | 147.33 | 348.03 | 59.33 | 140.16 |
| | 两次重复 | 72756 | 线性规划引导的贪心算法 | 31.00 | 67.00 | 216.13 | 33.00 | 106.45 |
| | 两次重复 | 72756 | ILP | 31.00 | 67.00 | 216.13 | 33.00 | 106.45 |
| | loop | 423 | 朴素贪心算法 | 48.67 | 49.00 | 100.68 | 50.33 | 103.42 |
| | loop | 423 | 线性规划引导的贪心算法 | 46.00 | 46.00 | 100.00 | 46.00 | 100.00 |
| | loop | 423 | ILP | 46.00 | 46.00 | 100.00 | 46.00 | 100.00 |
| | 三分 | 1358499 | 朴素贪心算法 | 155.00 | 149.33 | 96.34 | 169.33 | 109.25 |
| | 三分 | 1358499 | 线性规划引导的贪心算法 | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |
| | 三分 | 1358499 | ILP | 100.00 | 100.00 | 100.00 | 100.00 | 100.00 |

**表 8：采样错误修复的原始运行时间和运行时间比较**

| | | | | 基础流水线 | k = 2 | | k = 4 | |
|---|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | 运行时间 | 运行时间 | 运行时间缩减 | 运行时间 | 运行时间缩减 |
| Coreutils | 单向 | 45728 | 线性规划引导的贪心算法 | 113.9463 | 117.9029 | -3.4724 | 116.1778 | -1.9584 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 0.8512 | 0.8423 | 1.0449 | 0.8353 | 1.8685 |
| 立法 | 单向 | 166977 | 线性规划引导的贪心算法 | 111.9393 | 144.8334 | -29.3857 | 109.1641 | 2.4792 |
| | 双向 | 419439 | 线性规划引导的贪心算法 | | | | | |
| | loop | 3783 | 线性规划引导的贪心算法 | 0.2798 | 0.1843 | 34.1506 | 0.3096 | -10.6573 |
| ICIJ | 单向 | 558799 | 线性规划引导的贪心算法 | 183.4825 | 76.8191 | 58.1328 | 102.3670 | 44.2089 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 45.2682 | 34.3127 | 24.2014 | 83.7004 | -84.8988 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 2.3449 | 2.1603 | 7.8712 | 2.3681 | -0.9890 |
| | loop | 56614 | 线性规划引导的贪心算法 | 7.5130 | 4.7326 | 37.0083 | 5.4140 | 27.9375 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 25.9784 | 21.9824 | 15.3820 | 26.6876 | -2.7303 |
| LDBC | 单向 | 79064 | 线性规划引导的贪心算法 | 527.7366 | 524.6183 | 0.5909 | 518.9552 | 1.6640 |
| | 两次重复 | 72756 | 线性规划引导的贪心算法 | 3.3646 | 24.9187 | -640.6203 | 13.4611 | -300.0839 |
| | loop | 423 | 线性规划引导的贪心算法 | 0.0140 | 0.0167 | -19.1819 | 0.0177 | -25.9705 |

**表 9：采用采样错误进行修复时的原始边删除和删除比较**

| | | | | 基础流水线 | k = 2 | | k = 4 | |
|---|---|---|---|---|---|---|---|---|
| 数据集 | 形状 | 错误数 | 算法 | 删除 | 删除 | 百分比 | 删除 | 百分比 |
| Coreutils | 单向 | 45728 | 线性规划引导的贪心算法 | 6732.00 | 6810.33 | 101.16 | 6732.00 | 100.00 |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 603.00 | 603.00 | 100.00 | 603.00 | 100.00 |
| 立法 | 单向 | 166977 | 线性规划引导的贪心算法 | 2618.00 | 3367.33 | 128.62 | 2670.00 | 101.99 |
| | 双向 | 419439 | 线性规划引导的贪心算法 | | | | | |
| | loop | 3783 | 线性规划引导的贪心算法 | 38.00 | 218.00 | 573.68 | 92.67 | 243.86 |
| ICIJ | 单向 | 558799 | 线性规划引导的贪心算法 | 1188.33 | 1245.33 | 104.80 | 1204.33 | 101.35 |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 50.67 | 630.67 | 1244.74 | 207.67 | 409.87 |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 1176.33 | 1180.67 | 100.37 | 1175.67 | 99.94 |
| | loop | 56614 | 线性规划引导的贪心算法 | 725.00 | 762.67 | 105.20 | 740.33 | 102.11 |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 1.00 | 3.00 | 300.00 | 3.00 | 300.00 |
| LDBC | 单向 | 79064 | 线性规划引导的贪心算法 | 22438.00 | 22582.00 | 100.64 | 22443.00 | 100.02 |
| | loop | 423 | 线性规划引导的贪心算法 | 46.00 | 57.00 | 123.91 | 46.00 | 100.00 |
| | 两次重复 | 72756 | 线性规划引导的贪心算法 | 31.00 | 1239.33 | 3997.85 | 325.00 | 1048.39 |

**表10：Coreutils 和立法图的扩展性实验中的原始运行时间和运行时间比较（1 路约束）**

| 数据集和约束形状 | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|
| | 错误数 | 算法 | 运行时间 | 运行时间缩减 | 运行时间 | 运行时间增加 |
| Coreutils 单向 | 14809 | 朴素贪心算法 | 26.78 | 90.68 | 30.98 | 115.69 |
| | 14809 | 线性规划引导的贪心算法 | 10.78 | 96.24 | 12.37 | 114.76 |
| | 14809 | ILP | 287.15 | | 1245.41 | 433.72 |
| | 19109 | 朴素贪心算法 | 40.08 | 88.18 | 38.56 | 96.20 |
| | 19109 | 线性规划引导的贪心算法 | 12.19 | 96.41 | 10.48 | 86.01 |
| | 19109 | ILP | 339.057 | | 1401.39 | 413.32 |
| | 23393 | 朴素贪心算法 | 66.43 | 86.65 | 64.43 | 97.00 |
| | 23393 | 线性规划引导的贪心算法 | 35.27 | 92.91 | 22.94 | 65.05 |
| | 23393 | ILP | 497.72 | | 2078.88 | 417.68 |
| | 27661 | 朴素贪心算法 | 93.34 | 83.36 | 81.91 | 87.76 |
| | 27661 | 线性规划引导的贪心算法 | 43.98 | 92.16 | 22.92 | 52.12 |
| | 27661 | ILP | 561.02 | | 2352.06 | 419.25 |
| | 35101 | 朴素贪心算法 | 163.55 | 80.04 | 163.26 | 99.82 |
| | 35101 | 线性规划引导的贪心算法 | 56.32 | 93.13 | 52.72 | 93.61 |
| | 35101 | ILP | 819.24 | | 3224.69 | 393.62 |
| | 45728 | 朴素贪心算法 | 290.32 | 74.86 | 322.12 | 110.95 |
| | 45728 | 线性规划引导的贪心算法 | 110.73 | 90.41 | 105.31 | 95.11 |
| | 45728 | ILP | 1154.86 | | 超时 | |
| 立法 单向 | 26603 | 朴素贪心算法 | 11.52 | 84.53 | 17.28 | 150.00 |
| | 26603 | 线性规划引导的贪心算法 | 4.31 | 94.21 | 4.75 | 110.18 |
| | 26603 | ILP | 74.48 | | 309.06 | 414.95 |
| | 41047 | 朴素贪心算法 | 26.81 | 79.30 | 35.93 | 134.03 |
| | 41047 | 线性规划引导的贪心算法 | 9.81 | 92.43 | 8.29 | 84.55 |
| | 41047 | ILP | 129.51 | | 521.96 | 403.02 |
| | 91841 | 朴素贪心算法 | 57.60 | 69.23 | 85.64 | 148.67 |
| | 91841 | 线性规划引导的贪心算法 | 18.84 | 89.94 | 20.37 | 108.12 |
| | 91841 | ILP | 187.20 | | 766.74 | 409.57 |
| | 98686 | 朴素贪心算法 | 110.54 | 70.67 | 158.43 | 143.33 |
| | 98686 | 线性规划引导的贪心算法 | 43.41 | 88.48 | 48.60 | 111.97 |
| | 98686 | ILP | 376.91 | | 1692.88 | 449.15 |
| | 121968 | 朴素贪心算法 | 187.30 | 73.68 | 258.30 | 137.91 |
| | 121968 | 线性规划引导的贪心算法 | 61.78 | 91.32 | 65.68 | 106.31 |
| | 121968 | ILP | 711.57 | | 3079.11 | 432.72 |
| | 166977 | 朴素贪心算法 | 356.64 | 70.10 | 389.81 | 109.30 |
| | 166977 | 线性规划引导的贪心算法 | 109.13 | 90.85 | 98.02 | 89.82 |
| | 166977 | ILP | 1192.84 | | 超时 | |

**表 11：ICIJ 和 LDBC图扩展性实验的原始运行时间及运行时间比较（1路约束）**

| 数据集和约束形状 | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|
| | 错误数 | 算法 | 运行时间 | 运行时间缩减 | 运行时间 | 运行时间增加 |
| ICIJ 单向 | 1285 | 朴素贪心算法 | 0.027 | 87.97 | 0.03 | 100.03 |
| | 1285 | 线性规划引导的贪心算法 | 0.05 | 78.91 | 0.09 | 190.18 |
| | 1285 | ILP | 0.23 | | 0.54 | 235.78 |
| | 1517 | 朴素贪心算法 | 0.17 | 80.79 | 0.26 | 148.87 |
| | 1517 | 线性规划引导的贪心算法 | 0.19 | 79.20 | 0.25 | 135.68 |
| | 1517 | ILP | 0.89 | | 3.14 | 351.59 |
| | 2786 | 朴素贪心算法 | 0.42 | 94.35 | 0.58 | 138.17 |
| | 2786 | 线性规划引导的贪心算法 | 0.31 | 95.89 | 0.44 | 143.73 |
| | 2786 | ILP | 7.49 | | 28.53 | 380.64 |
| | 175724 | 朴素贪心算法 | 36.19 | -22.40 | 66.26 | 183.08 |
| | 175724 | 线性规划引导的贪心算法 | 24.13 | 18.41 | 52.79 | 218.80 |
| | 175724 | ILP | 29.57 | | 125.52 | 424.50 |
| | 558799 | 朴素贪心算法 | 324.12 | -445.65 | 559.79 | 172.71 |
| | 558799 | 线性规划引导的贪心算法 | 176.13 | -196.51 | 397.95 | 225.95 |
| | 558799 | ILP | 59.40 | | 294.69 | **496.11** |
| | 2052352 | 朴素贪心算法 | 超时 | | 超时 | |
| | 2052352 | 线性规划引导的贪心算法 | 超时 | | 超时 | |
| | 2052352 | ILP | 超时 | | 超时 | |
| LDBC 单向 | 8704 | 朴素贪心算法 | 14.2085507934292 | **97.29** | 25.46 | 179.22 |
| | 8704 | 线性规划引导的贪心算法 | 8.61 | 98.36 | 12.77 | 148.31 |
| | 8704 | ILP | 524.46 | | 2402.83 | 458.15 |
| | 15337 | 朴素贪心算法 | 54.99 | 97.12 | 99.98 | 181.80 |
| | 15337 | 线性规划引导的贪心算法 | 42.52 | 97.77 | 60.18 | 141.54 |
| | 15337 | ILP | 1909.62 | | 超时 | |
| | 55957 | 朴素贪心算法 | 143.65 | | 244.35 | 170.10 |
| | 55957 | 线性规划引导的贪心算法 | 69.12 | | 92.72 | 134.14 |
| | 55957 | ILP | 超时 | | 超时 | |
| | 79064 | 朴素贪心算法 | 980.82 | | 1627.58 | 165.94 |
| | 79064 | 线性规划引导的贪心算法 | 513.14 | | 847.47 | 165.15 |
| | 79064 | ILP | 超时 | | 超时 | |

**表 12：Coreutils 和立法图扩展性实验中的删除数量及比较次数原始值（1路约束）**

| 数据集和约束形状 | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|
| | 错误数 | 算法 | #边删除 | 删除量减少 | #标签删除 | 删除减少 |
| Coreutils 单向 | 14809 | 朴素贪心算法 | 3679 | | 2579 | 29.90 |
| | 14809 | 线性规划引导的贪心算法 | 2763.67 | 24.87 | 2092.00 | 24.30 |
| | 14809 | ILP | 2754 | 25.14 | 2092 | 24.04 |
| | 19109 | 朴素贪心算法 | 4115 | | 2655 | 35.49 |
| | 19109 | 线性规划引导的贪心算法 | 2848.33 | 30.79 | 2092.00 | 26.55 |
| | 19109 | ILP | 2824 | 31.38 | 2092 | 25.92 |
| | 23393 | 朴素贪心算法 | 5207 | | 3299 | 36.65 |
| | 23393 | 线性规划引导的贪心算法 | 4107.33 | 21.12 | 2586.67 | 37.02 |
| | 23393 | ILP | 4069 | 21.86 | 2586 | 36.45 |
| | 27661 | 朴素贪心算法 | 5753 | | 3344 | 41.87 |
| | 27661 | 线性规划引导的贪心算法 | 4346.00 | 24.46 | 2586.67 | 40.48 |
| | 27661 | ILP | 4257 | 26.00 | 2586 | 39.25 |
| | 35101 | 朴素贪心算法 | 6870 | | 4344 | 36.77 |
| | 35101 | 线性规划引导的贪心算法 | 5095.33 | 25.84 | 3375.00 | 33.76 |
| | 35101 | ILP | 5088 | 25.94 | 3375 | 33.67 |
| | 45728 | 朴素贪心算法 | 8747 | | 5657 | 35.33 |
| | 45728 | 线性规划引导的贪心算法 | 6732.00 | 23.04 | 4462.00 | 33.72 |
| | 45728 | ILP | 6732 | 23.04 | 超时 | |
| 立法 单向 | 26603 | 朴素贪心算法 | 1003 | | 630 | 37.21 |
| | 26603 | 线性规划引导的贪心算法 | 762.00 | 24.05 | 370.00 | 51.44 |
| | 26603 | ILP | 762 | 24.05 | 370 | 51.44 |
| | 41047 | 朴素贪心算法 | 1551 | | 760 | **50.98** |
| | 41047 | 线性规划引导的贪心算法 | 1082.00 | 30.24 | 441.00 | 59.24 |
| | 41047 | ILP | 1082 | 30.24 | 441 | **59.24** |
| | 91841 | 朴素贪心算法 | 1830 | | 1012 | 44.70 |
| | 91841 | 线性规划引导的贪心算法 | 1365.00 | 25.41 | 596.00 | 56.34 |
| | 91841 | ILP | 1365 | 25.41 | 596 | 56.34 |
| | 98686 | 朴素贪心算法 | 2421 | | 1390 | 42.59 |
| | 98686 | 线性规划引导的贪心算法 | 1797.00 | 25.76 | 774.00 | 56.93 |
| | 98686 | ILP | 1797 | 25.76 | 774 | 56.93 |
| | 121968 | 朴素贪心算法 | 3185 | | 1695 | 46.78 |
| | 121968 | 线性规划引导的贪心算法 | 2100.00 | 34.06 | 899.00 | 57.19 |
| | 121968 | ILP | 2100 | 34.06 | 899 | 57.19 |
| | 166977 | 朴素贪心算法 | 3819 | | 1905 | 50.10 |
| | 166977 | 线性规划引导的贪心算法 | 2618.00 | 31.44 | 1064.00 | 59.36 |
| | 166977 | ILP | 2618 | 31.44 | 超时 | |

**表 13：ICIJ与LDBC图扩展性实验中删除与比较的原始数量（1元约束）**

| 数据集与约束形状 | | | 基础流水线 | | 通过删除标签进行修复（步骤 2） | |
|---|---|---|---|---|---|---|
| | 错误数 | 算法 | #边删除 | 删除量减少 | #标签删除 | 删除减少 |
| ICIJ 单向 | 1285 | 朴素贪心算法 | 55 | | 39 | 28.05 |
| | 1285 | 线性规划引导的贪心算法 | 44.33 | 18.90 | 38.00 | 14.29 |
| | 1285 | ILP | 43 | 21.34 | 38 | 11.63 |
| | 1517 | 朴素贪心算法 | 186 | | 167 | 10.38 |
| | 1517 | 线性规划引导的贪心算法 | 176.00 | 5.55 | 149.33 | 15.15 |
| | 1517 | ILP | 174 | 6.62 | 149 | 14.37 |
| | 2786 | 朴素贪心算法 | 338 | | 261 | 22.88 |
| | 2786 | 线性规划引导的贪心算法 | 261.33 | 22.68 | 226.33 | 13.39 |
| | 2786 | ILP | 260 | 23.08 | 226 | 13.08 |
| | 175724 | 朴素贪心算法 | 753 | | 581 | 22.92 |
| | 175724 | 线性规划引导的贪心算法 | 627.67 | 16.68 | 476.00 | 24.16 |
| | 175724 | ILP | 616 | 18.23 | 463 | 24.84 |
| | 558799 | 朴素贪心算法 | 1411 | | 1008 | 28.54 |
| | 558799 | 线性规划引导的贪心算法 | 1188.00 | 15.80 | 826.00 | 30.47 |
| | 558799 | ILP | 1188 | 15.80 | 807 | 32.07 |
| | 2052352 | 朴素贪心算法 | 超时 | | 超时 | |
| | 2052352 | 线性规划引导的贪心算法 | 超时 | | 超时 | |
| | 2052352 | ILP | 超时 | | 超时 | |
| LDBC 单向 | 8704 | 朴素贪心算法 | 5782 | | 5441 | 5.89 |
| | 8704 | 线性规划引导的贪心算法 | 4854.00 | 16.045 | 4711.00 | 2.95 |
| | 8704 | ILP | 4854 | 16.04 | 4711 | 2.95 |
| | 15337 | 朴素贪心算法 | 11304 | | 10318 | 8.72 |
| | 15337 | 线性规划引导的贪心算法 | 10227.00 | 9.52 | 8978.00 | 12.21 |
| | 15337 | ILP | 10227 | 9.52 | 超时 | |
| | 55957 | 朴素贪心算法 | 12829 | | 11486 | 10.47 |
| | 55957 | 线性规划引导的贪心算法 | 11339.00 | 11.62 | 9732.00 | 14.17 |
| | 55957 | ILP | 超时 | | 超时 | |
| | 79064 | 朴素贪心算法 | 22438 | | 21099 | 5.97 |
| | 79064 | 线性规划引导的贪心算法 | 22438.00 | | 19740.00 | 12.02 |
| | 79064 | ILP | 超时 | | 超时 | |

**表 14：使用自定义权重（PageRank 分数）计算修复的运行时间**

| 数据集 | 形状 | 错误数 | 算法 | 运行时间 | 运行时间增加 | 边删除 | 删除量减少 |
|---|---|---|---|---|---|---|---|
| ICIJ | 单向 | 558799 | 朴素贪心算法 | 439.04 | -1655.31 | 2643.00 | |
| | 单向 | 558799 | 线性规划引导的贪心算法 | 185.26 | -640.69 | 1266.00 | 52.10 |
| | 单向 | 558799 | ILP | 25.01 | | 1264.00 | 52.18 |
| | 两次重复 | 397108 | 朴素贪心算法 | 212.25 | | 153.00 | |
| | 两次重复 | 397108 | 线性规划引导的贪心算法 | 40.14 | | 73.00 | 52.29 |
| | 两次重复 | 397108 | ILP | | | | |
| | 双向 | 7553 | 朴素贪心算法 | 4.30 | -693.78 | 1912.00 | |
| | 双向 | 7553 | 线性规划引导的贪心算法 | 2.43 | -349.08 | 1189.67 | 37.78 |
| | 双向 | 7553 | ILP | 0.54 | | 1188.00 | 37.87 |
| | loop | 56614 | 朴素贪心算法 | 7.32 | -312.05 | 732.00 | |
| | loop | 56614 | 线性规划引导的贪心算法 | 7.63 | -329.29 | 726.00 | 0.82 |
| | loop | 56614 | ILP | 1.78 | | 726.00 | 0.82 |
| | 三分 | 876933 | 朴素贪心算法 | 14.79 | 45.80 | 2.00 | |
| | 三分 | 876933 | 线性规划引导的贪心算法 | 26.10 | 4.39 | 1.00 | 50.0 |
| | 三分 | 876933 | ILP | 27.29 | | 1.00 | 50.0 |
| Coreutils | 单向 | 45728 | 朴素贪心算法 | 250.91 | -13373.86 | 8595.00 | |
| | 单向 | 45728 | 线性规划引导的贪心算法 | 103.88 | -5478.28 | 6737.00 | 21.62 |
| | 单向 | 45728 | ILP | 1.86 | | 6737.00 | 21.62 |
| | 双向 | 3707 | 朴素贪心算法 | 1.57 | -473.44 | 786.00 | |
| | 双向 | 3707 | 线性规划引导的贪心算法 | 1.13 | -310.75 | 751.00 | 4.45 |
| | 双向 | 3707 | ILP | 0.27 | | 708.00 | 9.92 |

**表 15：计算朴素贪心算法（NG）和线性规划引导的贪心算法（LPG）选择阶段近似修复的运行时间**

| 数据集 | 形状 | 错误数 | 算法 | 运行时间 |
|---|---|---|---|---|
| Coreutils | 单向 | 45728 | 近似天然气 | 0.38 |
| | 单向 | 45728 | 近似液化石油气 | 1.39 |
| | 双向 | 3707 | 近似 NG | 0.04 |
| | 双向 | 3707 | 近似 LPG | 0.10 |
| 立法 | 单向 | 166977 | 近似 NG | 1.69 |
| | 单向 | 166977 | 近似 LPG | 3.99 |
| | 双向 | 443695 | 近似 NG | 超时 |
| | 双向 | 443695 | 近似 LPG | 超时 |
| | loop | 3783 | 近似 NG | 0.14 |
| | loop | 3783 | 近似 LPG | 0.23 |
| ICIJ | 单向 | 558799 | 近似 NG | 11.15 |
| | 单向 | 558799 | 近似 LPG | 20.11 |
| | 两次重复 | 397108 | 近似 NG | 7.41 |
| | 两次重复 | 397108 | 近似 LPG | 19.72 |
| | 双向 | 7553 | 近似天然气 | 0.08 |
| | 双向 | 7553 | 近似 LPG | 0.22 |
| | loop | 56614 | 近似天然气 | 0.93 |
| | loop | 56614 | 近似 LPG | 1.68 |
| | 三分 | 876933 | 近似天然气 | 15.69 |
| | 三分 | 876933 | 近似液化石油气 | 26.00 |
| LDBC | 单向 | 79064 | 近似天然气 | 0.63 |
| | 单向 | 79064 | 近似 LPG | 2.77 |
| | 两次重复 | 72756 | 近似 NG | 1.32 |
| | 两次重复 | 72756 | 近似 LPG | 2.37 |
| | loop | 423 | 近似 NG | 0.00 |
| | loop | 423 | 近似 LPG | 0.01 |
| | 三分 | 1358499 | 近似 NG | 22.32 |
| | 三分 | 1358499 | 近似 LPG | 51.70 |

**表 16：朴素贪心算法（NG）和线性规划引导的贪心算法（LPG）的选择阶段所建议的近似修复中的边删除**

| 数据集 | 形状 | 错误数 | 算法 | 运行时间 |
|---|---|---|---|---|
| Coreutils | 单向 | 45728 | 近似天然气 | 11820.67 |
| | 单向 | 45728 | 近似液化石油气 | 6733.67 |
| | 双向 | 3707 | 近似天然气 | 744.33 |
| | 双向 | 3707 | 近似 LPG | 603 |
| 立法 | 单向 | 166977 | 近似天然气 | 5483.33 |
| | 单向 | 166977 | 近似 LPG | 2618 |
| | 双向 | 443695 | 近似 NG | |
| | 双向 | 443695 | 近似 LPG | |
| | loop | 3783 | 近似 NG | 87 |
| | loop | 3783 | 近似 LPG | 38 |
| ICIJ | 单向 | 558799 | 近似 NG | 1580.33 |
| | 单向 | 558799 | 近似 LPG | 1188 |
| | 两次重复 | 397108 | 近似天然气 | 113.67 |
| | 两次重复 | 397108 | 近似液化石油气 | 92 |
| | 双向 | 7553 | 近似天然气 | 1743.33 |
| | 双向 | 7553 | 近似 LPG | 1180 |
| | loop | 56614 | 近似 NG | 731.67 |
| | loop | 56614 | 近似 LPG | 725 |
| | 三分 | 876933 | 近似 NG | 9.67 |
| | 三分 | 876933 | 近似 液化石油气 | 1 |
| LDBC | 单向 | 79064 | 近似 天然气 | 30464.33 |
| | 单向 | 79064 | 近似 液化石油气 | 22438 |
| | 两次重复 | 72756 | 近似天然气 | 95.33 |
| | 两次重复 | 72756 | 近似液化石油气 | 31 |
| | loop | 423 | 近似天然气 | 51.33 |
| | loop | 423 | 近似液化石油气 | 46 |
| | 三分 | 1358499 | 近似 NG | 311.67 |
| | 三分 | 1358499 | 近似 LPG | 100 |