arXiv:2512.13999v1 [cs.DM] 16 Dec 2025

Misra 和 Gries 边着色算法的一个经过验证的实现

Arohee Bhoja

2025

摘要

Vizing 定理指出，每个简单无向图都可以用少于 $\Delta + 1$ 种颜色进行边着色，其中 Δ 是该图的最大度数。原始证明通过一个多项式时间算法过程给出，该过程迭代地扩展部分着色，直至其完整。在这项工作中，我使用 Lean定理证明器，对 Misra 和 Gries 边着色算法进行了验证实现。该算法是维辛原始方法的一个修改版本（源代码见 [1]）。本文重点构建相关数学对象的库，并严格维护所需的不变量。

1 引言

设 G 是一个简单无向图。我们用 $\Delta(G)$ 表示 G 的最大度数，或简写为 Δ。

- G 上的正常边着色将 G 中的边映射为颜色，使任意两条关联边的颜色都不相同。如果图中的每条边都被着色，则这种边着色称为完全。- G的边色数记作 $\chi'(G)$，它是对 G进行正常边着色所需的最少颜色数。

由此我们可以立即得出 $\Delta \leq \chi'(G)$。这是因为，如果两条边关联于同一个顶点，它们就不能使用相同的颜色；而顶点的度就是与其关联的边数。

1965年，苏联数学家瓦迪姆·维辛描述了一种多项式时间算法过程，并证明了 $\chi'(G) \leq \Delta + 1$。 [2] 事实证明，这是使用确定性多项式时间算法能够达到的最紧界，因为对于给定的图 G，判断 $\chi'(G) = \Delta$还是 $\chi'(G) = \Delta + 1$是NP完全问题。 [3]

20世纪后半叶，人们曾多次尝试简化 Vizing 最初提出的算法及其正确性证明。Bollob´as 于1979年提出了一个更为简短的证明，但由于省略了

论证中的若干非平凡部分，它实际上更像是一份概述。 [4] 20世纪90年代初，Rao 和 Dijkstra [5] 以及 Misra 和 Gries [6], 设计了更为简化的算法，引入了辅助数据结构的命名，并尽可能减少了分支。

然而，即使经过这些改进，正确性证明仍不可避免地相当复杂。每种算法的描述中都用到了若干辅助数据结构，这些结构以微妙的方式彼此交互。必须证明它们在多个子程序中都能维持某些不变量，而这些子程序本身也并非简单。换言之，算法本身并不能直接引出其正确性证明，因此它成为形式化验证中一个有趣的案例研究。

许多图论算法已使用交互式和自动定理证明器进行了形式化验证。2024年，Tobias Nipkow 在 Isabelle 中验证了 Gale–Shapley 稳定匹配算法，通过逐步细化，从抽象规约逐步推进到具体实现。 [7]然而，这种方法并不容易扩展到 Misra 和 Gries 算法，因为该算法的复杂性以及对辅助数据结构的依赖，使得逐步具体化难以实施。此外，2018年，Tarjan 算法（用于计算有向图的强连通分量）已在演绎式程序验证器 Why3 以及交互式定理证明器 Rocq 和 Isabelle 中得到验证。 [8] 尽管这些验证是有效的，但它们依赖于抽象库数据类型，因此并未构成对计算效率高的实现进行的严格证明。

Joachim Breitner 于 2011年早先在建模形式化语言 Event-B 中实现了 Misra 和 Gries 边着色算法，并使用 Rodin 工具对其进行了验证。 [9] Rodin 会自动从实现中提取证明义务，再由人工在 Isabelle 中完成证明。尽管最终得到的形式化正确性证明是可靠的，但仍有几项局限：它依赖于对 Rodin 可靠性的假设，而且自动生成的证明被形容为“写一次，永不再读”。本工作借助交互式定理证明器，力求通过建立非形式化证明与形式化证明之间更紧密的对应关系来缓解这些问题。

2 算法及其正确性证明

现在我们介绍该算法在形式化中的实现。
令 $V(G)$ 表示 G 的顶点集，令 $E(G)$ 表示其边集。
令 $u \in V(G)$，并令 c 为一种颜色。固定一个（部分）适当的 $(\Delta + 1)$-边着色 C。c 是一个:

- 关联颜色，当 u 上存在一条与 u 关联且着色为 c 的边时。- 可用颜色，否则为 u。

扇 $F = \langle f_1, \ldots f_k \rangle$ 顶点 x 的一组扇是由不同邻居组成的非空序列

满足以下性质的 x：

- 边 $\{x, f_1\}$ 未着色。- 扇着色性质：对于每个 $i < k$，$\{x, f_{i+1}\}$ 的颜色在 f_i上是空闲的。

如果从上下文中可以清楚地看出 x，则用扇中的元素作为其对应扇边的简称（例如，用 f_i 简称 $\{x, f_i\}$）。

如果我们想给 $\{x, f_1\}$着色，可以根据扇的性质轻松地完成这一过程——由于每条扇边的颜色在其前一条边上都是空闲的，我们可以将每种颜色沿边向下旋转一条边，并让最后一条扇边保持未着色。

对于每个 i，令 c_i 表示边 $\{x, f_i\}$的颜色。现在我们可以用伪代码描述这一过程：

> **函数** ROTATE $C[F]$对于每个 $i < k$，将 $\{x, f_i\}$ 重新着色为 c_{i+1}。取消 $\{x, f_k\}$的着色。

旋转引理。此操作可保持边着色的正确性。

证明。考虑任意一个 $i < k$。我们要证明 c_{i+1} 是 $\{x, f_i\}$ 的有效颜色（即它能保持边着色的正确性）。

根据定义，在扇旋转之前，c_{i+1} 在 f_i 上是空闲的。

此外，旋转之后，$\{x, f_i\}$ 必定是与 x 关联且颜色为 c_{i+1} 的唯一一条边。假设存在矛盾：有一个顶点 $v \neq f_i$ 与 x 关联，并且旋转之后 $\{x, v\}$ 被染成了 c_{i+1}。

情形 1：$v \in F$。那么，对某个 $j \neq i$，$v = f_j$，因此旋转之前 f_{j+1} 被染成了 c_{i+1}。但这是不可能的，因为 f_{i+1} 被染成了 c_{i+1}，而边着色是正确的。

情形 2：$v \notin F$。旋转只会影响扇中包含的边，因此旋转之前 $\{x, v\}$一定被染成了 c_{i+1}。我们还知道，旋转之前 f_{i+1} 被染成了 c_{i+1}，并且 $f_{i+1} \neq v$，因为 $v \notin F$。与前面一样，这是不可能的，因为边着色是正确的。

□

一条交替路径 $P = \langle p_1, \ldots p_l \rangle$ 在具有颜色 c，d的顶点 x 上，是一个由互不相同的顶点组成的非空序列，满足 $p_1 = x$，并且对于每个 $i < l$，$\{p_i, p_{i+1}\}$ 的颜色为 c 或 d。

如果这条路径是极大的（即无法向任一端延伸），我们可以反转这条路径，而不损害边着色的正确性。这可能有助于改变 x 上的可用颜色集合，因为如果在反转之前，c 是 x 上的可用颜色，那么反转之后 d 将是可用颜色，反之亦然。

我们用伪代码描述反转过程：

> **函数** INVERT $C[P]$

> 对于每个 $i < l$，翻转 $\{p_i, p_{i+1}\}$ 的颜色。如果它原本被着为 c，则将其重新着为 d；如果原本被着为 d，则将其重新着为 c。

反转引理。 如果 P 是极大的，此操作会保持边着色的有效性。

证明。 任取一个 $i < l$，不失一般性地假设 $\{p_i, p_{i+1}\}$ 被着成了 c。（另一种情况是对称的，可通过重新标记得出。）我们要证明，反转后 d 是 $\{p_i, p_{i+1}\}$ 的有效颜色。

由于边着色是正规的，我们知道：

1. 反转前，$\{p_{i-1}, p_i\}$ 和 $\{p_{i+1}, p_{i+2}\}$ 都被着成了 d。2. 与 p_i 或 p_{i+1} 相连的其他边都没有被着成 d。

反转后，$\{p_{i-1}, p_i\}$ 和 $\{p_{i+1}, p_{i+2}\}$ 都被着成了 c，因此 d 在 p_i 上空闲，而 p_{i+1} 也符合要求。 □

另一个有助于描述主算法的函数是 RE-COLOR，它会在边着色中设置一条边的颜色。

> **函数** RECOLOR $C[\{u, v\} \mapsto a]$返回由 C 给出的边着色，但将边 $\{u, v\}$ 重新着色为颜色 a

现在我们描述一个过程 EXTEND C U，它以部分（$\Delta + 1$）边着色 C 和集合 U 作为输入；集合 U 由 G 中在 C 下未着色的边组成。

选择 U 中一条边 $\{x, y\}$ 进行着色。主要思路是：如果我们能找到一个位于 x 上的扇，其第一个元素是 y，且最后一个元素与 x 共享某个可用颜色 u，那么只需旋转该扇，为 $\{x, y\}$着色，然后用 u为最后一条扇边着色。

接下来，我们的任务是找到这样的扇。我们从构造一个极大扇$F = \langle f_1, ..., f_k \rangle$ 开始，它位于 x 上，且第一个元素是 y。我们可以保证这样的扇存在，因为单元素扇 $\langle y \rangle$ 是有效的。然后，我们在x上选择一个可用颜色 c，并在 f_k上选择一个可用颜色 d。 c 和 d 必定存在，因为 G中的每个顶点至多有 Δ 种关联颜色，而 C 是一个（$\Delta + 1$）边着色。

如果 d在 x上是空闲的，我们可以旋转 F，用 d重新着色 $\{x, f_k\}$，然后递归地为 $U - \{x, y\}$中的其余边着色。
然而，如果 d 与 x关联，就必须使其空闲。我们通过反复选择颜色为 c 和 d 的边，在 x 上构造一条由颜色 c 和 d 组成的交替路径，直到路径无法再继续延伸。然后我们反转这条路径。由于反转前 c 在 x 上是空闲的，因此反转后 d 必定在 x 上空闲。

我们无法保证路径反演会保留 F 的性质，但可以证明，F 中存在一个具有所需性质的子扇——也就是说，存在某个 $w \leq k$，使得 $\langle f_1, ..., f_w \rangle$ 是一个扇，且 d 在 f_w 上是自由的。我们将这一事实称为**子扇存在引理**，并将在后文给出证明。

接下来，我们可以旋转这个子扇，使用 d 对 $\{x, f_w\}$ 重新着色，然后递归地为其余边着色。
以下是伪代码形式的算法：

1: 函数 EXTEND U C :=2: 如果 $U = \emptyset$ 那么 C3: 否则 4: 选择 $\{x, y\} \in U$。 5: 令 $F = \langle f_1, ...f_k \rangle$ 为 x 上的极大扇，满足 $f_1 = y$。 6: 令 c 为 x上的一个可用颜色，并令 d 为 f_k上的一个可用颜色。 7: 如果 d 是 x 上的一个关联颜色，那么 8: 令 P 为 x 上的一条极大交替路径，其颜色为 c, d。 9: $C :=$ INVERT $C[P]$10: 令 $w \leq k$ 满足 $F' = \langle f_1, ...f_w \rangle$ 构成一个扇，且 d 在 f_w上可用。 11: $C' :=$ ROTATE $C[F']$12: $C' :=$ RECOLOR $C'[\{x, f_w\} \mapsto d]$13: EXTEND $(U - \{x, y\})$ C'

如前所述，下面给出子扇存在引理的证明：

引理。交替路径的反转保证存在一个$w \leq k$，使得 $\langle f_1$、.....、 $f_w \rangle$ 构成一个扇，且 d在 f_w上是空闲的（参见： EXTEND的第 10 行）。

证明。若 d 在 x上是空闲的，取 $w = k$ 即可。否则：
注意， F必定恰有一条边着色为 d。它不可能多于一条，因为每个 f_i 都是 x 的邻居，而且 C 是正常边着色。它也不可能少于一条，否则我们可以将 x的邻居中关联边着色为 d 的那个追加到 F，这与 F的极大性矛盾。

令 $\{x, f_i\}$ 为这条边。
观察 *1*。 $i > 1$，因为 $f_1 = y$ 且 $\{x, y\}$ 未着色。
观察 *2.* 根据交替路径的定义， x 是 P的一个端点。由于 P 是极大的，它必定包含 x所有颜色为 c 或 d的邻居。因此我们可以得出结论： $f_i \in P$。

观察 *3.* d 在 f_{i-1}上是空闲的；根据扇的定义，这是因为 $\{x, f_i\}$ 被着色为 d。

情形 1： $f_{i-1} \notin P$。
路径反演只会影响着色为 c 和 d的边。 F 没有着色为 c 的边（因为 c 在 x上是空闲的）。因此，对于所有 $j \neq i$， f_j 上的空闲颜色以及 $\{x, f_j\}$ 的颜色都保持不变。由于 F 原本是一个扇，我们可以得出结论：对于所有 f_j，扇着色性质仍然成立。

取 $w = i - 1$。根据观察 3，我们就完成了。

情形 2： $f_{i-1} \in P$。取 $w = k$。
和之前一样，对于所有 $j \neq i$， f_j 都满足扇着色性质。根据观察 2，反转后 $\{x, f_i\}$ 被着色为 c。由于 d 在 f_{i-1} 上是空闲的（根据

观察 3），f_{i-1} 必须是 P 的一个端点，因此反转后 c 必须在 f_{i-1} 上空闲。

此外，$f_k \notin P$。假设事实如此；那么它就必须是一个端点，因为在反转之前，d 在 f_k 上是空闲的。然而，这是不可能的，因为x 和 f_{i-1} 是 P的两个端点。

因此，我们可以得出结论：d在 f_k上仍然空闲，因为反转并未影响它。

□

现在我们终于可以描述主算法 COLOR G了。它接收一个图，并返回一种完全正常边着色。

函数 COLOR G :=设 C 为一种空的（$\Delta + 1$）
边着色。EXTEND E（G） C

与包含副作用或可变数据的程序相比，纯函数式程序在数学上更容易推理。因此，COLOR与 Misra 和 Gries 提出的算法有所不同。

现在我们可以开始证明 Vizing 定理，它可由以下内容推出：

定理。 COLOR G 是 G 的完全正常边着色，且至多使用 $\Delta + 1$种颜色。

证明。 根据构造，EXTEND 只使用 $\Delta + 1$ 种颜色。我们刚刚已经论证过，在 EXTEND 中执行的每个操作都会保持其输入边着色的正确性。因此，我们可以应用归纳论证，得出结论：COLOR G 是正常边着色。

观察。 在 EXTEND的每次迭代开始时，$\{x, y\}$ 必须始终是未着色的。为说明这一点，假设 U 中的每条边在 C中都未着色。 G 中受 EXTEND 影响颜色的边，要么位于 P中，而其中不含未着色边；要么位于 F中，而其中唯一未着色的边是 $\{x, y\}$。因此，$U - \{x, y\}$ 中的每条边在 C'中都必须未着色。对此论证应用归纳法，即可得出所需结论。

现在我们可以通过另一个归纳论证来证明完备性。只需证明 C' 比 C多一条已着色的边。

反转路径不会影响已着色边的总数，旋转扇也不会（前提是 $\{x, y\}$ 未着色，这一点我们已在上文证明）。执行第 (6) 行后，$\{x, f_k\}$ 未着色；第 (7) 行将其着为 d，因此已着色边的总数按要求增加一条。

□

3 形式化：库

实现该算法所需的许多数学对象，在 Mathlib（Lean 的开源形式化数学库）中都有抽象定义。 [10, 11]然而，这些定义并未针对计算效率进行优化，因此我选择改用 Lean 核心数据类型，例如列表和数组。这使得该算法得以实现为多项式时间算法，但也需要投入大量额外工作——尤其是，我必须定义：

- 对每个数学对象进行表示，同时保留算法的多项式时间复杂度- 实现关键子程序（边重新着色、旋转扇和反转路径），并证明这些子程序能够保持其所修改对象的不变量- 对非形式化证明中未明确给出定义的各种概念（例如极大性）进行形式化

本节中的 Lean 代码经过了整理，并省略了一些技术细节，以便于阅读。

3.1 简单图

顶点表示为小于 n的自然数，边表示为顶点对。

```
variable(n :Nat)abbrev 顶点:=
Finnabbrev 边 := 顶点 n  ×顶点 n
```

类型 `Fin n` 表示小于 n的自然数。
邻居集合定义为一个由互不重复的顶点组成、长度小于n的列表；这里要求严格小于，以排除自环。（仅凭`nodupAx` 只能推出长度小于或等于 n，因此我们还施加了`sizeAx`这一附加条件。）

```
结构体 Nbors (n : 自然数)其中val :
List (顶点 n) sizeAx : val.length  <
nnodupAx :val.Nodup
```

这里，val.length 是 List.length val的简写。
谓词`Nodup`表示同一个顶点不能在邻居集合中多次存在——也就是说，不允许多重边。
简单图以邻接表表示，将顶点映射到其邻域。我们还要求满足以下额外性质：图中存在每个顶点，边是双向的，并且不存在自环。

结构 图（n：自然数）其中值：数组（Nbors n）大小Ax：值.大小 =n 对称Ax： ∀u v：顶点 n，u ∈ 值[v] ↔v ∈值[u]无自环Ax： ∀u：顶点 n，u ∉值[u]

值得注意的是，尽管图是无向的，边仍以元组表示。理想情况下，边类型应当是无序的，但 Mathlib 中无序对的类型（sym2）会让定理陈述变得更加繁琐，并且在形式化过程中需要使用许多额外的库引理。这个选择并没有带来太多额外工作——我定义了几个关于对称性的引理，并在必要时用它们改写陈述。不过，这或许是使用宏并进行重构的一个好机会。

图的边集定义为图的边组成的列表（即 $E(G$的子集），并满足以下几个附加性质：

- 如果 $(u,\ v)$ 存在于边集中，那么 $(v,\ u)$ 也必须存在。- 不允许有多重边。

3.2 边着色

3.2.1 定义

颜色定义为一种选项类型：要么是小于 c的自然数，要么是无。这样便可以表示部分边着色。
边着色 c G 是一种结构类型，其参数为待着色的图以及边着色中的颜色数量。数据以修改后的邻接矩阵表示，其中索引对应于一对边中存在的顶点，条目对应于颜色（若边未着色，则为none；若边以颜色 c 着色，则为some c）。

我们还有以下性质：

- 邻接矩阵的大小为 $n \times n$。- 边着色中已着色的每条边都存在于图中。- 任意两条关联边的颜色都不同（除非该颜色为 none ）。- 矩阵是对称的 —— C[u][v] = C[v][u]。

二维数组具有常数时间访问这一优良的计算特性，但在编写陈述和证明时，以及使用自动化来完成这些证明中的目标时，可能会带来困难。

例如，为了访问数组给定索引处的元素，Lean 要求提供一个证明，证明该索引在有效范围内。在二维数组中，这会变得尤其繁琐。下面是一个返回给定边的颜色的函数，位于

一种边着色：变量 (C :
EdgeColoringcG)

定义 颜色 (e : Edge n): =(C.val[e.1]'(通过 重写 [C.sizeAx1]; 精确 e.1.isLt))[e.2]'(通过 重写 [C.sizeAx2]; 精确e.2.isLt)

我发现，优先考虑计算效率，其次考虑自动化的易用性，再考虑陈述和证明的编写难易程度，是编写相对高效实现的最佳策略；这样一来，我就不必手动完成那么多困难且技术性强的目标。

Lean 文件中还有许多关于边着色的其他函数和引理，是我为便于证明后续定理而实现的。为简洁起见，此处省略。

3.2.2 实现重新着色操作

边着色库的最后一个重要部分，是实现上一节中的 RECOLOR 操作。

我们首先将有效性的概念形式化。如果颜色 a 对边 e有效（即边着色有效性 e a），那么 a要么是none，要么在路径P中的两个顶点上都未被占用。

我们实现一个函数，用于将二维矩阵中的单元格设为给定值。我们还证明以下规约：

- 数组的大小保持不变。- 数组中的所有其他值保持不变。- 指定单元格中的值确实是给定值。- 除第 i 行以外，数组中的每一行都保持不变。- 在二维方阵中设置两个不同的单元格是一种可交换操作。- 我们还包含一些引理，以便推理操作前后数组中给定值的计数。

许多列表和数组库函数的应用程序接口并不完整；我尽量避免使用这些函数。因此，库中包含了一个使用函数 边着色有效性谓词的规约，其中用到了 Array.count。这个引理指出，当且仅当 a 在 $\{u, v\}$ 上空闲时， $C[u]$和 $C[v]$ 中的 a 计数都为零。

现在，我们可以定义一个用于重新着色边的函数，它会设置边着色中与该边对应的两个单元格。此操作也有一组需要证明的规约。基本上，如果我们在边着色 C 中将边 $e = \{u, v\}$ 重新着色为颜色 a ：

- C 的大小和对称性得以保留。

- e 确实被赋予了 a，其他边的颜色保持不变。- 对于任意颜色 b，如果 $a = b$，则 u 和 v 的邻域中 b的计数增加一；如果 e 之前被赋予了 b，则计数减少一；否则保持不变。

最后，为了创建一个类型为 `EdgeColoring` 的对象，我们必须证明该操作保持边着色定义中列出的性质（大小、表示图、正确性和对称性）。这些证明篇幅较长且技术性较强，因为需要对索引进行分类讨论，并多次调用对称性引理；这里同样省略这些证明。

3.3 扇

3.3.1 定义

我们将顶点 x 和 y 上的扇 F 定义为一个非空且元素互不相同的数组，并满足以下附加性质：

- F 的第一个元素是 y。 - F 的每个元素都在 x 的邻域中。 - 扇着色性质如上一节所述成立。

为了实现扇着色性质，我们使用 Lean 中的 `Chain'` 函数，它表示给定谓词对列表中的相邻元素成立。这样我们便可以表达一个依赖于索引的性质，而无需实际使用索引，从而在之后省去许多证明义务。

```
def fan_prop (x : 顶点 n) (f₁ f₂ : 顶点 n):=colorC (x, f₂) ∈
freeColorsOnCf₁
```

```
def colorAx (F : 数组 (顶点 n)) (x : 顶点 n):=F.toList.Chain' (
扇_性质 C x)
```

我们还定义了顶点 x 和 y 上单元素扇的概念，并证明在图中存在边 $\{x, y\}$ 的假设下，扇性质成立。

3.3.2 创建极大扇

给定一条未着色的边 $\{x, y\}$，算法要求我们在 x 上生成一个极大扇，其第一个元素是 y。
我们首先定义一个函数 `add` ，给定一个部分扇 `F` 和一个邻居列表 `nbors`，这些邻居当前不在扇中；该函数会尽可能多地向扇中添加元素。

`add` 使用`find?`函数搜索Nbors，寻找一个顶点 z，使其颜色

$\{z, x\}$ 在F中的最后一个顶点上是空闲的。如果存在这样的 z ，则将其添加到F 的末尾，并从Nbors中删除。否则，原样返回F 。
如果传给List.erase 的值不在列表中，它会原样返回该列表。因此，要证明add会终止，我们必须证明在移除 z之前，它始终是nbors 的成员。默认情况下，在匹配语句的各个分支中，我们无法访问 z 的定义。因此，我们使用List.attach 函数，将 z 与证明它是 nbors成员的证据捆绑起来。这看起来不如在匹配判别式中添加一个假设那么简洁，但能让简化器和split 更有效地处理匹配语句。

def add (x y : 顶点 n) (F : 数组 (顶点 n)) (nbors : 列表 (顶点 n)) (h : F ≠# []) :=匹配 (List.attach nbors).find? (fun ⟨z, _⟩ ↦颜色 C (x, z) ∈freeColorsOn C (F.back (Array.size_pos_iff.mpr h))) with|some z => add x y (F.push z.val) (nbors.erase z.val) Array.push_ne_empty|none =>F 终止性_by nbors.length decreasing_by rw [List.length_erase_of_mem z.prop]apply Nat.sub_one_lt_of_le apply List.length_pos_iff_exists_mem.mpr exact ⟨z.val, z.prop⟩rfl

接下来，我们陈述并证明一些引理，说明扇性质在add操作下保持不变。由于add 不是结构递归的，我们使用fun_induction策略对函数定义本身进行归纳；尽管该函数结构复杂，这种方法仍能简化证明。

极大性条件的表述与函数定义本身类似——对扇调用add后， x中尚未存在于扇中的每个邻居，都不能在不违反着色不变式的情况下添加到扇的末尾。函数归纳使证明变得直接。

最后，我们定义一个包装函数maximalFan ，用于创建一个Fan类型的对象。对于数据，我们对单元素扇 $\langle y \rangle$ 和邻居集合nbhd x $-\{y\}$调用add 。对于性质，我们调用先前证明的关于add 的引理，以及单元素扇的性质。

3.3.3 实现旋转操作

要在边着色 C中用给定颜色 a旋转扇 $F = \langle f_1, ..., f_k \rangle$ ，我们将 $\{x, f_k\}$ 重新着色为 a，然后递归地用 $\{x, f_k\}$之前的颜色旋转扇 $\langle f_1, ..., f_{k-1} \rangle$ 。

由于扇的正式定义取决于边着色，我们

必须证明在 $C[\{x, f_k\} \mapsto a]$ 上，$\langle f_1, ..., f_{k-1} \rangle$ 仍然是一个扇，之后才能进行递归调用。

为此，我们编写一个函数 `mkFan`，它接收边着色 C 上的扇 $\langle f_1, ..., f_k \rangle$（以及颜色 a 和一些额外的假设），并返回边着色 $C[\{x, f_k\} \mapsto a]$ 上的扇 $\langle f_1, ..., f_{k-1} \rangle$。

定义 mkFan (a : 颜色 c) (hvalid : 边着色有效性 C (x, last F) a) (hsize : F.val.大小 `> 1)：Fan (setEdgeColor C (x, last F) a (last_`存在 F) hvalid) x y 其中 ...

接着，我们可以使用 `mkFan` 来实现上文所述的 `rotateFan` 操作。

定义 rotateFan (C : 边着色 c G) (F : 扇 C x y) (a : 颜色 c) (hvalid : 边着色有效性 C (x, last F) a) : 边着色 c G `:=`令 `a' :=`颜色 C (x, last F)令 `C' :=` `setEdgeColor C(x, last F) a (last_`存在 F) hvalid如果 h : F.val.大小 `> 1` 那么令 `F' :=mkFanFahvalidh`有 hvalid' : 边着色有效性 C' (x, last F') a' `:=` 通过 ...`rotateFanC' F'a'hvalid'`否则 C' 终止性_通过 ... 递减_通过 ...

3.4 路径

3.4.1 定义交替性质

在陈述交替路径的定义之前，我们必须定义一种性质：它会在列表中连续的值之间交替。遗憾的是，`Chain'`在这里并不适用，因为要描述交替这一特性，我们需要访问每个元素的索引。

相反，我们递归地定义一个函数 `alternates`，它接收一个函数p、两个值 a, b和一个列表 vs。`alternates p a b vs` 表示函数p 在对列表中连续的元素求值时，其值会在 a 和 b 之间交替。也就是说，如果 $vs = \langle v_1, v_2, v_3, ..., v_n \rangle$，$p\ v_1\ v_2 = a$, $p\ v_2\ v_3 = b$，依此类推。

定义 alternates {V C：类型*} (p：V →V →C) (a b：C)：列表 V →命题
`| [] =>`真`| _ :: [] =>`真`| v₁ :: (v₂ :: vs) =>p v₁ v₂ = a ∧ alternatespba(v₂ ::vs)`

我们还定义了一个函数`next`，它会返回在列表末尾追加一个符合交替性质的元素时，p 应取的下一个值

性质。也就是说，对于任意元素 w，`alternates p a b (vs ++ [w])` 当且仅当`next a b vs` $= p\ v_n\ w$。

定义 `next {V C : 类型*} (ab : C) : 列表 V → C`
`| [] =>b| _ ::vs =>nextbavs`

我们证明这些函数的若干性质，以便之后更轻松地对它们进行推理。以下是几个例子，其中 $vs = \langle v_1, ..., v_n \rangle$ 是一个任意列表。

- next a b vs 总是等于 a或 b。- 对于某个元素 w，如果 alternates p a b vs 且 $p\ v_n\ w$，alternates p a b (vs ++ [w])。- 如果 alternates p a b (x :: xs)，那么 alternates p b a xs。

以两种不同的方式看待一个列表通常很有用：将其视为归纳定义，或视为一个把索引映射到值的函数。在 Lean 中这是可行的，但同时采用这两种视角并不特别自然，而且在不同表示之间转换往往需要大量工作。为减少今后的繁琐，我们证明几个涉及索引的引理：

- 对于任意列表 L，如果 `next a b L = a`，那么 $p\ L[L.length - 2]\ L[L.length - 1] = b$，反之亦然（我们将使用相互归纳来证明）。- 设 $0 < i < L.length - 1$。那么，$p\ L[i-1]\ L[i] = a$ 和 $p\ L[i]\ L[i+1] = b$，反之亦然。

此外，还有几个关于 `Chain'` 的库引理，可以在这些视角之间进行转换。这些引理之后可能会派上用场，因此我们证明：如果 `alternates p a b L`，那么列表链 (函数 `x y ↦ p x y = a ∨ p x y = b`) `L`。

3.4.2 构造一条极大路径

到目前为止，我们将每个数学对象都定义为包含数据和不变量的结构类型。这种做法使代码组织得更好、可读性更强，但也带来了一个不利影响：Lean 的许多自动化策略（包括归纳，而本节中的许多证明都需要用到归纳）因此效果会变差。

为了解决这个问题，我们现在将数据和不变量声明为变量，证明一些必要的引理，然后在后文给出每个引理的“简洁版本”。

变量
(P : List (顶点 n)) (a b : 颜色 c) (hne : P ≠ []) (x : 顶点 n)
(hx : P[0]'(...) = x)(hfree :b ∈
freeColorsOnCx)(hnodup :P.Nodup)(ha :a.isSome)(hb :
b.isSome)(hneq: a ≠ b)

这里，路径表示为一个顶点列表，其中相邻元素之间的边被着色为 a或 b。我们要求 a 和 b 是实际颜色（而不是 `none`），并且路径不包含环。与非正式定义一样， x 是路径的第一个元素。`hfree` 不是路径不变量，但我们可以在此处假设它成立，并在算法中创建极大路径时再补充这一条件。

算法要求我们构造一条极大的交替路径，其中包含顶点 x ，且其边的颜色为 a 或 b，因此我们的下一项任务是实现这一点。

为方便起见，我们使用 `alternates` 定义谓词 `alternatesColor` 。我们还定义函数 `nextVertex`，用于查找一个可能添加到交替路径中的下一个候选顶点。结果必须通过一条颜色为 a 或 b 的边与路径中的最后一个顶点相连。如果这样的顶点 z 存在，`nextVertex` 将返回 `some z`；否则，它将返回 `none`。

为了之后证明极大性，我们还必须证明下一个顶点的任何候选顶点都不会已经存在于路径中。非形式化证明从未论证这一点，而要将其形式化既复杂又繁琐（篇幅几乎长达 75 行！）。以下是证明概要：

设 $P = \langle p_0 = x,...,p_n \rangle$.。令 `some z` 为 `nextVertex` 返回的值。为导出矛盾，假设 z 存在于 P 中。那么，必定存在某个 i 使得 $z = p_i$。

情形 1： $i = 0$，因此 $z = p_0 = x$。由于 P 是一条交替路径， $\{p_n, x\}$ 必须着成 a 或 b。b 在 x 上是空闲的，因此我们可以得出 $\{p_n, x\}$ 着成了 a，而 $\{p_{n-1}, p_n\}$ 必须着成 b。

由于边着色是正常的， x 不能有两条关联边分别着成 a和 $p_n = p_1$。 P 不能包含重复项，因此 $P = \langle x, p_1 \rangle$ 和 $p_{n-1} = x$。我们得出矛盾，因为 $\{x, p_1\}$ 不可能同时着成 a 和 b。

情形 2： $0 < i < n$。由此我们立即知道 $n > 1$。
由于 P 是一条交替路径， $\{p_n, p_i\}$ 必须着成 a 或 b。不失一般性，假设 $\{p_n, p_i\}$ 着成 a。由于 p_i 是一个内部顶点，我们有 $\{p_{i-1}, p_i\}$ 着成 a，或者 $\{p_i, p_{i+1}\}$ 着成 a。在前一种情况下，$n = i - 1$，但这是不可能的，因为 $i < n$。因此，假设 $\{p_i, p_{i+1}\}$ 着成 a。那么， $n = i + 1$，并且由于最后一条路径边着成了 a，下一条必须着成 b。我们得出矛盾，因为 $\{p_n, p_i\}$ 不可能同时着成 a 和 b。

情况3： $i = n$，所以 $p_i = p_n$。 $\{p_n, p_i\}$ 是一个自环，因此必须取消着色。但 $\{p_n, p_i\}$ 必须着成 a 或 b中的一种颜色，而两者都不能是无，因此我们得出了矛盾。

有了这个引理，我们就可以定义一个函数，将现有路径不断 extend，直到它达到极大。`extendPath` 的结构与扇库中的 `add` 相似（也就是说，它不是结构递归的，因此我们必须手动证明其终止性）。我们首先搜索一个要添加到路径中的候选顶点 z；如果不存在这样的顶点，就原样返回该路径。否则，我们将 z 添加到路径中，并证明

在对所得路径递归调用 `extendPath` 之前，仍然满足每个路径不变量。

```
def extendPath {a b : Color c} {x : 顶点 n} (路径P : 列表 (顶点 n)) (hne : 路
径P ≠ []) (hx : 路径P[0]'(by exact List.length_pos_iff.mpr hne) =x)
(hfree : b ∈ freeColorsOn C x) (hnodup : 路径P.Nodup) (ha : a.isSome)
(hb : b.isSome) (hneq : a ≠b) (hcolor : alternatesColor C 路径P a b) : 列表
(顶点 n) :=match Option.attach (nextVertex C 路径P a b hne) with|none =>
路径P| some ⟨z, h⟩ => have hnodup' : (路径P.concat z).Nodup := by . . .
have hcolor' : alternatesColor C (路径P.concat z) a b := by . .. have hx' : (路
径P.concat z)[0]'(by simp) = x := by . . . extendPath (路径P.concat z) (by
simp) hx' hfree hnodup' ha hb hneq hcolor' 终止性_by (n + 1) - 路径
P.length decreasing_by . . .
```

和之前一样，我们使用函数归纳证明路径不变量得以保持，并证明该函数会生成一条极大的路径。此处省略证明。

3.4.3 定义

现在终于可以将交替路径定义为一种结构类型：

```
结构 路径P (a b : 颜色 c) (x : 顶点 n) 其中val : List (顶点
n) nonemptyAx : val ≠ []firstElemAx :val[0] =
xnodupAx :val.NodupcolorAx :alternatesColorCvalab
```

构造极大路径的策略与构造扇的版本类似：我们对单元素路径调用 `extend` 。我们定义一个用于刻画路径极大性的谓词（即路径中的下一个颜色在其末端顶点上是空闲的），并证明极大路径满足这一条件。

3.4.4 推理列表中的相邻元素

本节前面我提到过，在证明有关列表的定理时，一个常见的挑战是：很难在归纳地看待列表与将列表视为把索引映射到值的函数这两种视角之间转换。对于列表中相邻对的推理，这一点尤为明显，因为 Mathlib 尚未定义相邻对这一概念。我们将在这里定义它。

首先，给出一个基于索引的定义：如果元素 u 和 v 在列表 xs 中相邻，那么必定存在一个索引 i，使得 $u = xs[i]$ 和 $v = xs[i+1]$，或者反过来。

变量
{α :类型*} {xs :
列表 α}

定义 相邻 (u v : α) (xs : 列表 α) :=
(∃ i, ∃ (h : i <xs.length - 1), xs[i] =u ∧xs[i+1] =v) ∨(∃ i, ∃ (h : i <
xs.length - 1), xs[i] =v ∧xs[i+1] =u)

其次，归纳定义：给定一个列表 xs，所有相邻对 xs 返回 xs 中每个相邻对组成的列表。

定义 所有相邻对 : 列表 α →列表 (α × α)| [] => [] [_] => []x_1 :: x_2 ::
xs => (x_1 ,x_2) :: (x_2 ,x_1) :: (所有相邻对 (x_2 ::xs))

我们证明一个引理，使我们能够在这些相邻概念之间进行转换。这里函数归纳很有用，因为递归情形并不对应于列表的归纳结构。

定理 mem_所有相邻对_当且仅当_相邻 (u v: α) (xs: 列表 α):(u, v) ∈ 所有相邻对 xs ↔相邻 u v xs: =

3.4.5 路径边

根据定义，(u, v) 是路径边，当且仅当 u和 v在 P中相邻。因此，我们将 P中的相邻对定义为路径边：

定义 路径边: =所有相邻对 路径P.va l

我们再证明几个有用的性质：

- 令 u, v在 P中相邻，并令 v为一个内部顶点。必定存在某个与 v相邻的顶点 w，使得(u, v)和(u, w)的颜色不同。 - 如果顶点 v 属于一条极大路径，那么它的任何颜色为 a 或 b 的关联边都必须存在于该路径中。

3.4.6 实现求逆操作

我们最后要做的工作是实现前面描述的 INVERT 操作。

最简单的解决方案是依次翻转每条边的颜色。遗憾的是，这种方法在这里行不通，因为我们将 setEdgeColor 定义为只允许将边重新着色为有效颜色。只要巧妙地选择假设，我们

就可以递归地翻转路径的尾部，然后重新着色头部。不过，由于正确性证明需要我们对多个嵌套函数调用进行推理，我选择了一种不那么优雅但更简单的方案——先取消整条路径的着色，然后按相反的顺序重新着色。这个中间步骤让证明不变量得以维持变得更容易。

我们实现一个取消着色函数，该函数接收一个顶点列表，并递归地将每个相邻对的边颜色设置为none。我们还证明了一些辅助引理：

- 设置非路径边的颜色不会影响路径的颜色不变量。- 如果某种颜色在取消路径着色之前对某个顶点是空闲的，那么之后也必定是空闲的。- 取消路径着色后， a 和 b在每个路径顶点上都必须是空闲的。- 每条非路径边都不受取消路径着色的影响。- 取消路径着色后，路径中的每条边都被着色为none。

现在我们可以开始定义重新着色，该函数接收一个边着色 C、颜色 a、 b和一个顶点列表 L，以及一些假设。如果 L 是空列表或单元素列表，我们将边着色原样返回。否则，令$L = p_1 :: p_2 :: ps$。我们对边着色 $C[(p_1、 p_2) \mapsto b]$、颜色 b、a （顺序反转）以及顶点列表 $p_2 :: ps$递归调用重新着色 。

这里唯一微妙之处在于，我们无法在 L中的每个顶点上始终假设a 和 b 都是空闲的。在过程一开始，这确实成立；但一旦给第一条路径边着色， b在 p_2上就不再空闲。我们通过更谨慎地选择假设来处理这种不对称性：我们始终可以保证，最后使用的颜色在 L的尾部元素上是空闲的，而接下来要使用的颜色在 L的每个元素上都是空闲的。要证明每次setEdgeColor 操作都有效，这些条件就足够了。

def recolor (C : 边着色 c G) (a b : 颜色 c) (ha : a.isSome) (hb : b.isSome) (hne : a $\neq$ b) (L : 列表 (顶点 n)) (hnodup : L.Nodup) (h1 : 列表链 (fun e_1 e_2 $\mapsto$存在 G (e_1, e_2)) L) (h2 : $\forall$ v $\in$ L.尾部, a $\in$ freeColorsOn C v) (h3 : $\forall$ v $\in$ L, b $\in$ freeColorsOn C v) : 边着色 c G :=match hl : L with| [] => C| [_] => C| p_1 :: p_2 :: ps => have hpres : 存在 G (p_1, p_2) := . . . have hvalid : 边着色有效性 C (p_1, p_2) b := . . .have auxa : $\forall$ v $\in$ p_2 :: ps, a $\in$ freeColorsOn (setEdgeColor C (p_1, p_2) b hpres hvalid) v := . . .have auxb : $\forall$v $\in$(p_2 :: ps).尾部,

b ∈ freeColorsOn (setEdgeColor C (p₁, p₂) b hpres hvalid) v := ...重新着色 (setEdgeColor C (p₁, p₂) b hpres hvalid) b a hb ha (...) (p₂ :: ps) (...) (...) auxb auxa

我们证明以下规约：

- 如果一条边不在路径中，它的颜色就保持不变。- 路径在 b 和 a 两种颜色之间交替。

主要的反转函数 `invert` 会取消路径的着色，然后再为其重新着色。

定义 反转 {路径P : Path C a b x} (h : 极大路径性质 路径P) : EdgeColoring c G := 重新着色 (取消着色 C a b ha hb 路径P.val 路径P.nodupAx 路径P.colorAx) a b ha hb hne 路径P.val 路径P.nodupAx (...) (...)

为了证明其正确性，我们先定义谓词 `isInverted`。该谓词接收两个边着色 C 和 C'，以及一条路径 P。`isInverted C C' P` 表示对于所有边：

- isInverted_不属于 C C' 路径P：若 $e \notin P$，color C e = color C' e。- isInverted_属于 C C' 路径P：若 $e \in P$：– 若color C e = a，则color C' e = b。– 若color C e = b，则color C' e = a。

最后，我们证明更多关于反转的规范引理：

- 反转 函数返回的边着色满足isInverted谓词。- 若一条边在反转前未着色为 a或 b，则反转不会影响它。- 若一条边在反转后未着色为 a 或 b，则它在反转过程中未受影响。- 若一个顶点存在于极大路径中，且反转后有一条关联边着色为 a 或 b，则该边必定存在于路径中。- 若 a 在反转前对某个路径顶点是空闲的，则反转后 b必定仍为空闲（反之亦然）。

为方便后续证明的撰写，这些引理中有许多只是彼此的变体。

4 形式化：算法

该算法的实现与非形式化证明中的伪代码相似：我们定义一个子程序 `extendColoring`，它接收一个正常边着色，并递归地对其进行扩展，直到着色完整。主函数mkEdgeColoring会对空边着色调用extendColoring。

几点说明：

- 为简洁起见，省略了每个 陈述的证明。- 要证明终止性，必须证明 E 中的每条边在 C 中都已取消着色。这一点通过归纳证明，与非形式化版本类似。- 我们将在下一节实现 `findSubfan`，并证明子扇存在引理。

下面是两个函数的实现；它们与非形式化证明中的伪代码并没有太大差别！

定义 `extendColoring(E:EdgeSetG)(C:EdgeColoring(maxDegreeG + 1)G)(h : ∀e ∈ E.val,colorCe =none):EdgeColoring (maxDegree G + 1) G :=匹配 hE : E.val 其中| [] => C(x,y):: es =>有hterm :(removeE(x,y)).val.length < E.val.length有h': ∀ e ∈(removeE(x,y)).val,colorCe =none`

令 `F : =maximalFan C (...)有auxF: isMaximalF`

令 `a: = (freeColorsOnC(lastF)).head(existsFreeColorC(由by omega) (last F))令 b: = (freeColorsOnCx).head(existsFreeColorC(由 by omega) x)有 ha :a.isSome有hb :b.isSome有auxa: a ∈freeColorsOnC(lastF)有auxb: b ∈ freeColorsOnCx`

如果heq: a =b则有aux: 边着色有效性 C (x, last F) a令C' : = (rotateFanCFaaux)有h': ∀ e ∈(移除 E (x, y)).val, color C' e = none extendColoring (移除 E (x, y)) (rotateFan C F a aux) h'

else
令 P: 路径 C a b x: =极大路径 C ha hb heq (by simp [b]) 有 auxP: 极大路径性质 P 令 C': =反转 ha hb heq auxb auxP 有 auxC': isInverted C C' P 令 F': =findSubfan ha hb auxb auxC' auxa (by simp_all) auxF auxP 有 aux: 边着色有效性 C' (x, last F') a 令 C'': =(rotateFan C' F' a aux) 有 h': ∀ e ∈ (移除 E (x, y)).val, color C'' e =none extendColoring (移除 E (x, y)) C'' h'

终止性_由
E.val.length

定义 mkEdgeColoring（G : 图 n）: 边着色（最大度数 G + 1） G: =

extendColoring (toEdgeSet G) empty (...)

4.1 寻找子扇

交替路径的反转保证存在 F 的一个子扇，使得 a 在其最后一个元素上是自由的。

子扇存在引理的非形式化证明较为微妙，因此要完成形式化，我们必须证明以下命题：

- 如果一个顶点不是路径端点，那么路径反演不会影响它的空闲颜色。 - 如果某种颜色不等于 a 或 b，且在路径反演之前是某个顶点的可用颜色，那么反演之后它仍然是可用颜色。 - 如果路径为空，则扇的颜色性质得以保留。（这显然成立，因为反演不会产生任何变化，但要将其形式化仍需一些工作。） - 假设存在一条颜色为 a 的扇边 e 。那么，扇中除 e 以外的每条边都不会因路径反演而改变。 - 如果不存在颜色为 a 的扇边，那么包含 x 的极大路径就是一个单元素列表。

此处省略了引理的陈述和证明，因为其结构与对应的非形式化版本相同，但需要大量的分类讨论和算术。控制结构（尤其是能让我同时处理多个目标的策略）以及专用自动化工具（例如 `omega` 和 `grind`）尤其有帮助。

既然我们已经证明了子扇的存在性，就可以继续寻找它了。这项任务很直接——如果存在某个索引 i 使得 $\{x, f_i\}$ 被着色为a且为 $f_{i-1} \notin P$，我们就返回 F 的前 $i-1$ 个元素。否则，返回F 不变。在 Lean 中：

```
定义 mkSubfan (F : Fan C xy) (P : 路径 C a b x) : 数组 (顶点 n) : =匹配
Array.findFinIdx? (函数 u ↦ color C (x, u) = a) F.val 与|somei => 如
果F.val[i.val- 1] ∈P.val则F.val否则 Array.extractF.val0 i| none =>
F.val
```

现在我们可以编写这个函数的一个“精巧”版本，使其返回一个扇对象（即包含每个扇性质的证明）。剩下需要证明的最后一个性质是， a 始终在子扇的最后一个顶点处空闲。为简洁起见，此处省略证明。

4.2 正确性证明

在非形式化证明中，正确性论证包含两个主要部分：正确性和完备性。第一部分已经完成——mkEdgeColoring会生成一个边着色类型的对象，而根据定义，它是正确的！剩下的就是证明完备性。

这里采用一种常见的策略——首先，我们证明 extendColoring 能够保持正确性（也就是说，边集中每条被移除的边在图中都有颜色）。

```
定理 extendColoring_
proper(E:EdgeSetG)(C:EdgeColoring(maxDegree G + 1)G)(he1:  ∀e
 ∈ E.val,colorCe =none)(e :Edgen) (h1:e ∈(toEdgeSetG).val)(h2 :
e  ∉E.val →(colorCe).isSome):(color
(extendColoringEChe1)e).isSome:=
```

证明与非正式版本类似（但需要用到我们之前证明的旋转引理），因此这里省略。

现在可以陈述最后一个正确性证明：mkEdgeColoring 会生成一个正常边着色。

```
定理 mkEdgeColoring_proper:∀e  ∈
(toEdgeSetG).val,(color(mkEdgeColoringG)e).isSome:=
```

该证明几乎可以直接由前一个引理得出，至此证明完成。

5 结论

在这项工作中，我使用 Lean 4 证明助手编写了 Misra 和 Gries 边着色算法的多项式时间验证实现。该算法最初是为了简化一个复杂定理的证明而开发的，因此，拥有一个经过计算机验证的正确性证明令人振奋！

使用交互式定理证明器至关重要——许多证明都需要精妙的论证，完全自动化的工具很难综合出这些论证。不过，这一选择也带来了一系列挑战：

- 数学证明中常见的一种策略，是在同一对象的不同表示之间进行转换。例如，在一个证明中，我经常需要既以归纳方式看待列表，又将其视为一个把索引映射到值的函数。在形式化证明中，这类转换更具挑战性；为此，我编写了专门的引理，在需要时完成这些转换。

- 将数学算法形式化会引入一些相互竞争的目标，在实现过程中必须对它们进行权衡：便于从数学角度推理的表示

并不总是计算效率最高的，反之亦然。我选择优先考虑计算效率，因为这是表示本身固有的属性，若不改变底层表示，就无法改进。为处理由此产生的繁杂复杂的证明，我采用了多种策略，尤其是在可能的情况下借助自动化。

- 这些考量引出了一个问题：计算和证明是否采用完全独立的表示会更容易，然后再证明它们等价。我猜想这样证明会更简单，尤其是如果使用 Mathlib 中的定义；不过，这需要进行大量额外的双向转换。

未来的工作有许多可能方向。这个库有助于实现更多具有良好计算特性的、经过验证的图论算法。此外，许多地方若采用宏、自定义自动化，或选择不同的表示方式，证明本可以更简短、更优雅。随着这类工作的不断推进，我很期待看到表示和推理数学对象的最佳实践逐渐形成。

参考文献

[1] Arohee Bhoja。《Misra 和 Gries 边着色算法的经过验证的实现》。URL: https://github.com/aroheebhoja/vizing.[2] V. G. Vizing。“具有给定色类的临界图”。载于：*MetodyDiskret. Analiz.* 5（1965），第 9–17 页。[3] Ian Holyer。“边着色的 NP 完全性”。载于：《*SIAM* 计算杂志》10.4（1981），第 718–720 页。DOI: 10.1137/0210055。电子预印本：https://doi.org/10.1137/0210055。URL: https://doi.org/10.1137/0210055.[4] B´ela Bollob´as。《图论：入门课程》。第 63 卷。Springer Science & Business Media，2012 年。[5] Josyula R. Rao 和 Edsger W. Dijkstra。“Vizing 定理证明的设计”。载于：《编程与数学方法》。Manfred Broy 编。柏林、海德堡：Springer Berlin Heidelberg，1992 年，第 17–25 页。ISBN: 978-3-642-77572-7。[6] J. Misra 和 David Gries。“Vizing 定理的构造性证明”。载于：《信息处理快报》41.3（1992），第 131–133 页。ISSN:0020-0190。DOI: https://doi.org/10.1016/0020-0190(92)90041-S。URL: https: // www.sciencedirect.com/science/article/pii/002001909290041S。[7] Tobias Nipkow。“Gale-Shapley 验证”。载于：《自动演绎杂志》68.2（2024 年 6 月），第 12 页。ISSN: 1573-0670。DOI: 10.1007/s10817-024-09700-x。URL: https://doi.org/10.1007/s10817-024-09700-x。

[8] Ran Chen 等。“在 Why3、Coq 和 Isabelle 中对 Tarjan 算法的形式化证明”。载于：*CoRR* abs/1810.11979（2018 年）。arXiv：1810.11979。URL: http://arxiv.org/abs/1810.11979。[9] Joachim Breitner。《使用 Rodin 证明 Vizing 定理》。2011 年 3 月。URL: https://www.joachim-breitner.de/。[10] Leonardo de Moura 和 Sebastian Ullrich。“Lean 4 定理证明器与编程语言”。载于：《自动演绎——*CADE 28*》。Andr´e Platzer 和 Geoff Sutcliffe 编。Cham：Springer International Publishing，2021 年，第 625–635 页。ISBN: 978-3-030-79876-5。[11] mathlib 社区。“Lean 数学库”。载于：《第九届 *ACM SIGPLAN* 国际认证程序与证明大会论文集》。CPP 2020。美国路易斯安那州新奥尔良：计算机协会，2020 年，第 367–381 页。ISBN: 9781450370974。DOI: 10.1145/3372885.3373824。URL: https://doi.org/10.1145/3372885. 3373824。