图着色（Graph Coloring）算法

杨其臻

Aug 01, 2025

图着色（Graph Coloring）问题在计算机科学和日常生活中扮演着至关重要的角色。它源于一个直观的挑战：如何用最少的颜色为图中的顶点着色，确保相邻顶点颜色不同。这种问题看似简单，却隐藏着深刻的计算复杂性。例如，在地图着色场景中，相邻国家需用不同颜色以避免混淆；在课程排表中，冲突的课程需分配到不同时间段；在编译器设计中，寄存器分配要求共享资源的变量不能同时激活。这些实际应用突显了图着色在资源优化和冲突避免中的核心价值。本文的目标是系统性地引导读者理解经典图着色算法的思想，亲手实现代码，并分析优化策略。我们将从基础理论入手，逐步过渡到实践编码，最终探讨实际应用和前沿方向，帮助读者建立全面认知。

1　基础概念与问题建模

在深入算法前，我们需要回顾图论基础并形式化定义问题。一个图由顶点（Vertex）和边（Edge）组成，其中边表示顶点间的邻接关系。图着色问题的核心是寻找一个合法着色方案，即分配颜色函数 $C: V \rightarrow \{1,2,\ldots,k\}$，使得对于任意边 $(u,v) \in E$，有 $C(u) \neq C(v)$。关键术语包括色数 $\chi(G)$，它代表图 G 所需的最小颜色数；冲突指相邻顶点颜色相同；合法着色则确保无冲突。问题形式化为：输入一个无向图 $G=(V,E)$，输出最小 k 和对应的颜色分配。然而，图着色是 NP-完全问题，这意味着精确求解在大规模图中不可行，因为时间复杂度可能指数级增长，迫使我们依赖启发式或近似算法。理解这一特性有助于后续算法选择，避免在工程实践中陷入计算瓶颈。

2　贪心算法

贪心算法是图着色中最直观的求解方法，其核心思想是逐顶点着色，始终选择当前可用的最小颜色编号。具体步骤包括：首先对顶点进行排序，排序策略直接影响结果质量；接着遍历每个顶点，检查其邻居已使用的颜色集合；然后分配最小可用颜色。时间复杂度为 $O(V^2 + E)$，其中 V 是顶点数，E 是边数，这源于邻居检查的双重循环。贪心算法的主要缺陷在于结果依赖于顶点顺序：如果低度顶点优先着色，可能导致高阶顶点冲突增多，增加所需颜色数。例如，一个随机排序的图可能使用更多颜色，而优化排序能显著改善性能。尽管简单高效，但贪心法不保证最优解，仅提供可行方案。

3　威尔士-鲍威尔算法

威尔士-鲍威尔算法（Welsh-Powell）是对贪心法的优化，通过按顶点度数降序排序来提升着色效果。其执行流程分为三步：计算所有顶点度数；按度数从大到小排序；然后应用贪心着色策略，优先处理高优先级顶点。高优

先级顶点先着色能减少高阶顶点的冲突概率，因为它们有更多邻居，早期分配避免颜色耗尽。以下 Python 代码演示了这一实现，使用邻接表表示图：

```
def welsh_powell(graph):
    degrees = [(v, len(neighbors)) for v, neighbors in graph.items()]
    degrees.sort(key=lambda x: x[1], reverse=True) # 按度数降序排序
    color_map = {}
    for vertex, _ in degrees:
        used_colors = {color_map[n] for n in graph[vertex] if n in color_map} # 收集邻居
            ↪ 已用颜色
        for color in range(len(graph)): # 尝试从最小颜色开始分配
            if color not in used_colors:
                color_map[vertex] = color
                break
    return color_map
```

代码解读：函数 welsh_powell 接收一个字典 graph，其中键为顶点，值为邻居列表。第一行计算每个顶点的度数，存储为元组列表；第二行使用 sort 方法按度数降序排序，reverse=True 确保高度数顶点优先。接着初始化 color_map 字典存储着色结果。循环遍历排序后的顶点：对于每个顶点，通过集合推导式 used_colors 收集邻居已用颜色，避免冲突；内层循环从 0 开始尝试颜色，一旦找到可用颜色（color not in used_colors），就分配给当前顶点并跳出循环。返回的 color_map 包含合法着色方案。时间复杂度仍为 (O(V^2 + E))，但优化排序通常降低实际颜色数。例如，在一个环图中，威尔士-鲍威尔法比随机贪心少用 20% 的颜色。

4 回溯法

回溯法适用于小规模图的精确求解，目标是找到最小色数 (\chi(G))。其核心是递归框架：状态包括当前顶点索引和部分颜色分配；在每次递归中，尝试为顶点分配颜色，并检查合法性；如果冲突发生，则回溯撤销选择。剪枝策略是关键，例如提前终止无效分支：当部分解已出现冲突时，跳过后续递归。时间复杂度最坏为 (O(k^V))，其中 (k) 是颜色数上限，(V) 是顶点数，呈指数级增长，因此仅适合顶点数少的图。回溯法能保证最优解，但计算开销大，需权衡精确性和效率。

5 高级算法简介

除了基础算法，高级方法如 DSATUR 和递归最大优先（RLF）提供了更优性能。DSATUR 动态选择饱和度最高顶点着色，饱和度定义为邻居已用颜色数，能自适应调整顺序；RLF 则分批次着色独立集，减少冲突。这些算法虽复杂，但在大规模图中提升效率，例如 DSATUR 的时间复杂度接近 (O(V \log V))。

我们将使用 Python 实现经典算法，因其易读性和广泛库支持。图表示采用邻接表，即以字典存储顶点和邻居列表，例如 graph = {'A': ['B','C'], 'B': ['A'], 'C': ['A']} 表示一个三角形图。这比邻接矩阵更节省空间，尤其对于稀疏图。在威尔士-鲍威尔算法实现中，关键点包括度数计算和颜色分配逻辑。代码中 used_colors 使用集合高效检查邻居颜色，避免线性扫描；颜色尝试从 0 开始，确保最小化颜色编号。可视化输出可通过 networkx 和 matplotlib 库实现，例如绘制顶点颜色分布图，帮助直观验证算法正确性。实验时

建议从小图开始，如 5-10 个顶点，逐步扩展到复杂网络。

图着色在大规模图中面临性能瓶颈，如回溯法的指数级爆炸或贪心法的顺序依赖。优化技巧包括预着色策略：固定部分顶点颜色以缩小搜索空间；或颜色交换策略（Kempe Chains），通过交换冲突顶点的颜色链修复局部方案。算法选择需基于问题规模和要求：对于小型图且需最优解，推荐回溯法；对于快速可行解，威尔士-鲍威尔法更高效；若无需最优，启发式算法如 DSATUR 是首选。决策流程可描述为：根据问题规模，若小型则选回溯法，否则选威尔士-鲍威尔；根据最优性需求，若需最优则回溯法，否则贪心或启发式。工程中常结合多种策略，例如预着色后应用贪心法。

图着色算法在多个领域展现价值。在编译器设计中，寄存器分配问题将变量视为顶点，冲突使用视为边，着色确保寄存器高效复用；无线通信的频率分配中，基站为顶点，干扰为边，着色避免信号冲突；排课系统中课程为顶点，时间冲突为边，着色生成无冲突课表。这些案例证明算法的实用性，例如寄存器分配减少 CPU 空闲时间，提升程序性能。

探索方向包括特殊图的色数结论：如二分图 (\chi = 2)，平面图受四色定理约束 (\chi \leq 4)；并行图着色利用 GPU 加速，如 CuGraph 库实现分布式计算；现代算法库如 NetworkX 提供内置着色函数，可对比性能。这些方向推动算法创新，例如 GPU 并行处理百万级顶点图，大幅缩短求解时间。

图着色问题没有“银弹算法”，需根据场景权衡最优性和速度。贪心法快速但非最优，回溯法精确但昂贵，启发式如威尔士-鲍威尔提供平衡。鼓励读者动手实验，通过代码实现和可视化加深理解，例如修改顶点排序观察颜色变化。在实践中，结合理论深度与工程技巧，能有效解决冲突分配问题，推动技术创新。