

**HUBEIUNIVERSITY OF AUTOMOTIVE TECHNOLOGY**



**算法设计与分析**

**实 验 报 告**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 实验项目 | 实验四 | 实验类别 | | | 验证性 |
| 学生姓名 | 章崇文 | 学生学号 | | | 202202296 |
| 学生班级 | 计算机222 | | | | |
| 完成日期 | 2024-06-03 | | | | |
| 实验成绩 |  | | 评阅日期 |  | |
| 评阅教师 |  | | | | |

**实验四：****回溯法**

**【实验目的】**

应用回溯法求解图的着色问题

**【实验性质】**

综合性实验

**【实验要求】**

设下图G=(V,E)是一连通无向图，有3种颜色，用这些颜色为G的各顶点着色，每个顶点着一种颜色，且相邻顶点颜色不同。试用回溯法设计一个算法，找出所有可能满足上述条件的着色法，如果这个图不能用3种颜色着色满足相邻顶点颜色互异的要求就给出否定的回答。

**【算法思想及处理过程】**

**回溯法基本思想**

回溯法是一种通过探索所有可能的候选解来寻找所有解决方案或任一解决方案的算法。其核心在于“尝试-放弃”策略：算法尝试分步构建解，当发现当前路径无法得到有效解时，会撤销上一步或几步的操作，即“回溯”，然后尝试其他可能的分支。这一过程不断重复，直到找到所有解或满足停止条件为止。

**处理该问题的详细算法步骤**

定义数据结构：

使用邻接表adjList存储图G的结构，键为顶点编号，值为与该顶点相邻的所有顶点列表。

vector<int> colors用于记录当前的着色状态，其中colors[i]表示第i个顶点的颜色。

初始化：

设置颜色种类数m=3。

初始化计数器sum记录可行的着色方案数。

为每个顶点初始化颜色为0，表示未着色。

编写isValidColoring函数：

检查给定节点的颜色是否合法，即检查与之相邻的所有顶点是否有相同颜色。

实现回溯函数backtrack：

终止条件：当所有顶点都被着色（即node==colors.size()），增加解计数，并输出当前的着色方案。

递归过程：对于当前节点，尝试每一种颜色（1到m），如果颜色合法，则分配给当前节点，并递归到下一个节点；若不合法或所有颜色都已尝试，则回溯（重置当前节点颜色）并返回上一层。

主函数执行流程：

构建图的邻接表结构。

调用backtrack(0, colors)，从图的第一个顶点开始进行着色尝试。

最后，输出总共有多少种有效的着色方案。

**【程序代码】**

#include <iostream>

#include <vector>

#include <unordered\_map>

using namespace std;

unordered\_map<int, vector<int>> adjList;

int m = 3; // 假设色数为3

int sum = 0; // 符合条件的着色方案数量

char getColorCode(int color)

{

switch (color)

{

case 1:

return 'A';

case 2:

return 'B';

case 3:

return 'C';

default:

return '?';

}

}

// 判断当前点的着色是否合法

bool isValidColoring(vector<int> &colors, int node, int color)

{

for (int neighbor : adjList[node])

{

if (colors[neighbor] == color)

return false;

}

return true;

}

// 回溯函数

void backtrack(int node, vector<int> &colors)

{

if (node == colors.size())

{

sum++;

// 打印当前的着色方案

for (int i = 0; i < colors.size(); ++i)

{

cout << getColorCode(colors[i]);

}

cout << endl;

return;

}

for (int c = 1; c <= m; ++c)

{

if (isValidColoring(colors, node, c))

{

colors[node] = c;

backtrack(node + 1, colors);

colors[node] = 0; // 回溯，重置当前节点颜色

}

}

}

int main()

{

// 直接初始化图结构

adjList = {

{1, {2, 4, 0}}, // 注意调整索引以匹配点的实际编号

{2, {1, 3, 0}},

{3, {2, 4}},

{4, {1, 3, 0}},

{0, {1, 2, 4}} // 确保所有点都被初始化

};

int pointnum = adjList.size();

vector<int> colors(pointnum, 0);

backtrack(0, colors); // 从第一个点（索引为0）开始搜索

cout << "一共有 " << sum << " 种绘色方案" << endl;

return 0;

}

**【运行结果】**



**【算法分析】**

时间复杂度

回溯过程

在回溯过程中，对于每一个顶点，我们都尝试用m种颜色进行着色。在一个包含n个顶点的图中，理论上每个顶点都需要检查m种颜色的可能性。然而，由于回溯算法的剪枝特性（即当发现某条路径不可行时会立即回溯，避免了不必要的探索），实际尝试的组合会少于n\*m。

最好情况: 如果图很容易着色，比如是一棵树或者分层明显的图，每个顶点的选择都会很快排除不适合的颜色，算法会很高效。尽管如此，至少需要遍历所有顶点，所以时间复杂度不会低于O(n)。

最坏情况: 在完全图（每个顶点与其他所有顶点都相连）的情况下，每次选择颜色时都需要检查所有相邻顶点的颜色，这会导致大量的冲突检查。此时，每个顶点的着色尝试都会进行m次，且随着递归深入，会有指数级的增长。因此，在最坏情况下，时间复杂度接近O(n^m)。

综合考虑，算法的平均时间复杂度为O(n\*m^n)，其中n是顶点数，m是颜色数。这是因为对于每个顶点，都需要尝试m种颜色，并且整个过程呈树状展开，每一层对应一个顶点的着色决策。

空间复杂度

递归栈: 空间复杂度主要由递归深度决定，最大递归深度为n（即图中的顶点数），因为需要存储到目前为止的决策路径。因此，递归栈的空间复杂度是O(n)。

辅助数据结构: 除了递归栈外，还需要存储图的邻接表或邻接矩阵，以及当前的着色状态数组colors。邻接表或邻接矩阵的空间复杂度取决于图的边数和顶点数，但通常是O(n+m)，其中m是边的数量。colors数组占用O(n)空间。

综上所述，该算法的空间复杂度为O(n+m)，主要是存储图结构和当前着色状态所需的内存。

【**实验总结**】