深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验名称：回溯法—地图填色问题

学院：计算机与软件学院 专业：计科

报告人：欧阳宇杰 学号：2021150143 班级：计科 2 班

同组人：欧阳宇杰

指导教师：杜智华

实验时间：2023 年 3 月 28 日~2023 年 4 月 15 日

实验报告提交时间：2023 年 4 月 18 日星期二

教务处制

一、实验目的

（1）掌握回溯法算法设计思想

（2）掌握地图填色问题的回溯法解法。

二、实验背景与要求

背景知识：

为地图或其他由不同区域组成的图形着色时，相邻国家/地区不能使用相同的颜色。 我们可能还想使用尽可能少的不同颜色进行填涂。一些简单的“地图”例如棋盘仅需要两种颜色，但是大多数复杂的地图需要更多颜色。

每张地图包含四个相互连接的国家时，它们至少需要四种颜色。1852 年，植物学专业的学生弗朗西斯·古思里于 1852 年首次提出“四色问题”。他观察到四种颜色似乎足以满足他尝试的任何地图填色问题，但他无法找到适用于所有地图的证明。这个问题被称为四色问题。长期以来，数学家无法证明四种颜色就够了，或者无法找到需要四种以上颜色的地图。直到 1976 年德国数学家沃尔夫冈·哈肯和肯尼斯·阿佩尔使用计算机证明了四色定理，他们将无数种可能的地图缩减为 1936 种特殊情况，每种情况都由一台计算机进行了总计超过 1000 个小时的检查。他们因此工作获得了美国数学学会富尔克森奖。在 1990 年，哈肯成为伊利诺伊大学高级研究中心的成员，他现在是该大学的名誉教授。

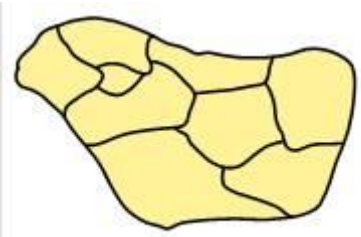
四色定理是第一个使用计算机证明的著名数学定理，此后变得越来越普遍，争议也越来越小 更快的计算机和更高效的算法意味着今天您可以在几个小时内在笔记本电脑上证明四种颜色定理。

问题描述：

将地图转换为平面图，每个地区变成一个节点，相邻地区用边连接，为这个图形的顶点着色，并且两个顶点通过边连接时必须具有不同的颜色。附件是给出的地图数据，请针对三个地图数据尝试分别使用 5 个（le450*5a*），*15* 个（*le450*15b），25 个（le450_25a）颜色为地图着色。

实验要求：

1、对下面这个小规模数据，利用四色填色测试算法的正确性；



2、对附件中给定的地图数据填涂；

3、随机产生不同规模的图，分析算法效率与图规模的关系（四色）。

三、实验步骤与结果

❖ 关于地图数据读取

顶点定义如下，记录了该顶点的所选颜色、对于每种颜色该顶点是否可选、可选颜色的数量和相邻点的数量即顶点的度。

注：0 代表没颜色，颜色从 1 开始，COLOR 为颜色总数。

```
struct Vertex
{
    int color; //该点所选的颜色
    int state[COLOR + 1];  //颜色状态，1 为可选，非 1 为不可选
    int choice;  //可选颜色的数量，即 state 中 1 的数量
    int degree;  //相邻点的数量,即该点的度
    Vertex() {
        color = 0; //默认该顶点没颜色
        for (int i = 0; i <= COLOR; ++i)
            state[i] = 1; //默认初始化为全部颜色都可选
        choice = COLOR; //默认初始化为全部颜色都可选
        degree = 0;  //默认初始化度为 0
    }
};
```

我通过邻接表来储存每个节点的相邻节点信息，包括当前节点的相邻节点总数（也就是度）和相邻节点具体是哪些（即记录当前节点的所有相邻节点编号）。

```
int Map[VERTEX + 1][256];
//邻接表,Map[i][0]表示第 i 个节点相邻节点的数量，Map[i][j]表示第 i 个节点的第 j 个相邻节点。
```

❖ 回溯法基础版本

回溯法的基本思想是从问题的初始状态出发，逐步地尝试所有可能的解决方案，当发现当前的解决方案不符合要求时，就回溯到之前的状态，尝试其他可能的解决方案，直到找到符合要求的解决方案或者已经尝试了所有的解决方案。

在地图填色问题中，回溯法具体应用如下：对当前进行填色时，如果颜色合法，则填下一个节点。如果所填颜色非法，则不继续搜索，而是回溯到上一层的节点填别的颜色。

大致描述了我想法的伪代码如下:

```
1.  void DFS(current,count){
2.      if(count==点总数){
3.          sum+=S[current].剩余可选颜色数 //到达叶子节点,统计结果
4.      }
5.      for(i = 1 to 颜色总数){
6.          if(当前颜色 i 可选){
7.              S[current].color=i;
8.          }
9.          if(check(current,i)){ //检查当前节点填颜色 i 是否合法
10.             DFS(下一个节点)
11.           //合法则 DFS 继续搜索后再回溯,不合法就不继续搜索了直接回溯
12.         }
13.         回溯
14.     }
15. }
```

❖ 探索如何剪枝

➢ 剪枝策略 1：向前探测—check 函数的实现

在上面的伪代码中我通过 check 函数其实做到了一种剪枝策略,那么这个 check 函数要如何实现呢?

check 函数即在填涂当前节点颜色时，对当前节点的相邻待填色节点进行遍历，若此节点填当前颜色 i 后会导致其相邻待填色节点中有节点无色可填，那么表明当前节点填颜色 i 是不可行的，是没有前途的。所以我们可以对当前节点填颜色 i 这种情况不继续向下探索，而是直接回溯，这样就做到了剪枝。

下面是一个示例（可供选择的颜色为红黄蓝绿 4 种），观察下图可以发现，在当前已填涂三个区域的情况下，将进行填涂第四个区域的操作，该区域剩余两种颜色可供选择。若选择红色，那么对该区域的相邻节点进行遍历未出现可选颜色为 0 的情况，因此保留；若选择蓝色，那么该区域的相邻节点中就有节点（即圆形区域）会出现可选颜色为 0 的情况，因此对此进行剪枝。

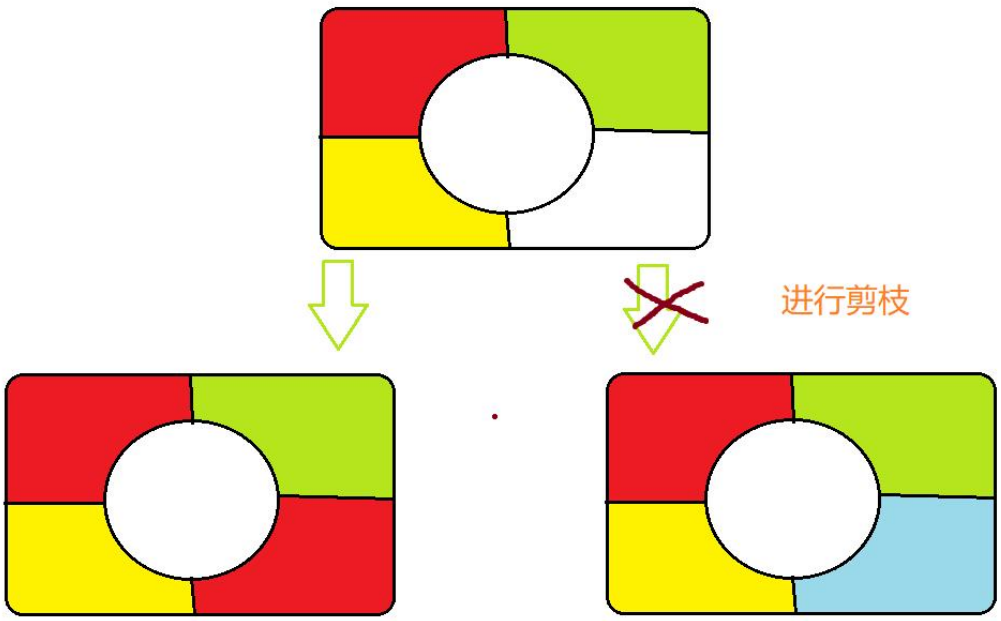


图 1 – 剪枝策略–向前探查法分析例子

根据上面的思路实现的 check 函数具体代码如下：

```
1.  bool check(Vertex* S,int current,int i){
2.      //传进节点数组、当前节点序号 current 和其所选的颜色 i
3.      for (int k = 1; k <= Map[current][0]; ++k) { //遍历当前节点的所有相邻节点
4.          int j = Map[current][k]; //j 是遍历到的相邻节点序号
5.          if (S[j].color == 0 && S[j].state[i] == 1) {
6.              //若顶点 j 为待填色节点且可以填颜色 i
7.              S[j].state[i] = -current; //就使得顶点 j 不能选 i 这个颜色
8.              S[j].choice--; //顶点 j 的可选颜色数量-1
9.              if (!S[j].choice ) {
10.                 //若顶点 j 的可选颜色数量-1 之后变为了 0,返回 false
11.                 return false;
12.             }
13.         }
14.     }
15.     return true; //若不会导致出现相邻顶点无色可选的情况则返回 true
16. }
```

➢ 剪枝策略 2：MRV+DH—选点策略的实现

上面描述 DFS 函数的大致伪代码的 9-12 行如下:

```
if(check(current,i)){ //检查当前节点填颜色 i 是否合法
    DFS(下一个节点)
 //合法则 DFS 继续搜索后再回溯,不合法就不继续搜索了直接回溯
}
```

我在剪枝策略 1 中给出了 check 函数的大致实现，那么如果 check 函数检查合格，程序就要从未填色的点中找到下一个要着色的点进行着色，所以如何挑选下一个需要着色的顶点就是一个值得思考的问题。在阅读了网上相关资料和老师课件中有关地图填色的内容后，我选择使用 MRV 最少剩余量选择策略 + DH 最大度选择策略，其中优先使用 MRH，其次 DH。

1. MRV(最少剩余量选择)：在待填色的节点中寻找可选颜色数最小的节点。
2. DH(度最大选择)：在可选颜色数最小的节点中选择度最大的作为下一个需要着色的节点。

为什么实现了剪枝？

MRH：优先填涂可选颜色数最少的节点，即先对容易导致失败的节点填涂，那么便可以早些发现是否当前填色方案必然失败，从而可以提前回溯，避开不少的不可行解。

DH：优先对度最多的节点进行填色，假设填色为 c，因为度最多的节点约束了**最多的**其他节点，使得**最多的**其他节点都不会去探索节点填色为 c 的情况（因为不合法），使得**最多的**其他节点的填色为 c 的分枝被剪掉了，从而实现了剪枝。

选点函数实现的具体代码：

```
int getNext(Vertex* S) { //优先 MRH,其次 DH
    auto Min = COLOR;//最小的可选颜色数量,初始化为最大
    int next = 0; //记录下一个需要着色的节点的序号
    for (int i = 1; i <= VERTEX; ++i) {
        if (!S[i].color ) { //从未填色的点中找到下一个要着色的点
            if (S[i].choice == Min) {
          //在可选颜色数最小的节点中选择度最大的作为下一个需要着色的节点
                if (S[i].degree > S[next].degree) {
                    Min = S[i].choice;
                    next = i;
                }
            }
            else if (S[i].choice < Min) {//寻找可选颜色数最小的节点
                Min = S[i].choice;
                next = i;
            }
        }
    }
    return next; //返回下一个需要着色的节点的序号
}
```

❖ 通过小规模数据测试算法正确性

算法优化剪枝至此，我对实验要求给出的小规模数据，利用四色填色测试算法的正确性。

注：以下所有运行时间均是我在自己的计算机上在VS2022的Release模式下跑出的。

1) 对于该小地图，首先我对各个区域进行编号如下，并将区域间的邻接关系记录于smallMapExample.txt文档中。

```
e 1 2
e 1 3
e 1 4
e 2 3
e 2 4
e 2 5
e 3 4
e 4 5
e 4 6
e 4 7
e 5 6
e 5 8
e 6 7
e 6 8
e 6 9
e 7 9
e 8 9
```

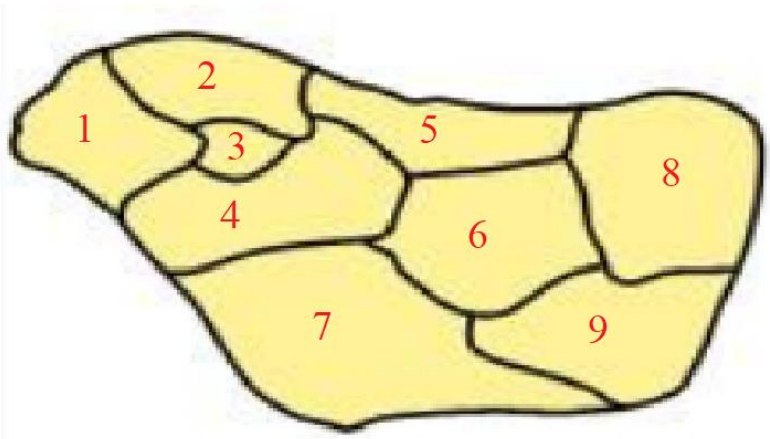
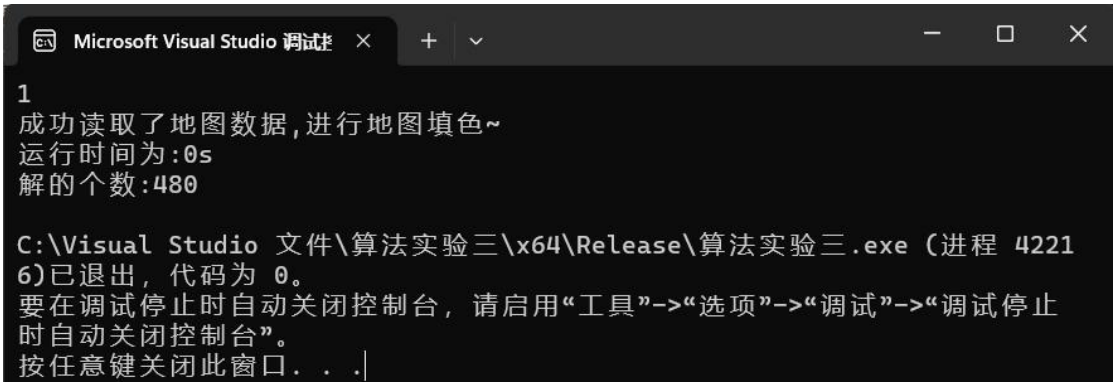


图2 - 对小地图各个区域编号并记录区域间的邻接关系

2) 我利用四色填色测试了算法的正确性，从下图可以看到对于这个小地图，程序在<0.5ms内找到了全部480个解，解的个数正确，算法是正确的。



通过上面的对小地图的测试,我们知道该算法是正确的的，那么该算法是否有继续优化改进的空间？

- **剪枝策略 3：我称为“树层去重”**

前提：每个节点填色时都按颜色编号从小到大尝试填涂(从 1 到 COLOR)

我在阅读相关资料和自己思考后，得到：对于回溯树形结构的同层节点(也即对应于图的同一个节点填不同颜色)，一旦某节点填涂了一个之前节点都没用过的新颜色 c，我们假设由该节点填涂该新颜色 c 不断拓展最后得到的可行解的数量为 n，那么该节点填涂其他新颜色(序号>c 序号)不断拓展最后得到的可行解的数量也为 n，故我们可以利用这个已知剪去大量的分支。

具体操作是一旦同层节点的某节点填涂了一个之前节点都没用过的新颜色，那么我

们在不断深搜计算得到由该节点填涂该新颜色不断拓展最后得到的可行解的数量 n 后，就不再去深搜计算该节点填其他新颜色时的解的数量，因为该节点填其他新颜色时解的数量也为 n，既然解的个数是一样的，那么就算一次就足够了。统计解总个数的变量 sum += n×新颜色总数。我们对该树形结构的每一层都做这样的剪枝操作，那么可以想象我们将减去该树形结构大部分的分枝。

我将这种剪枝策略称为“树层去重”。

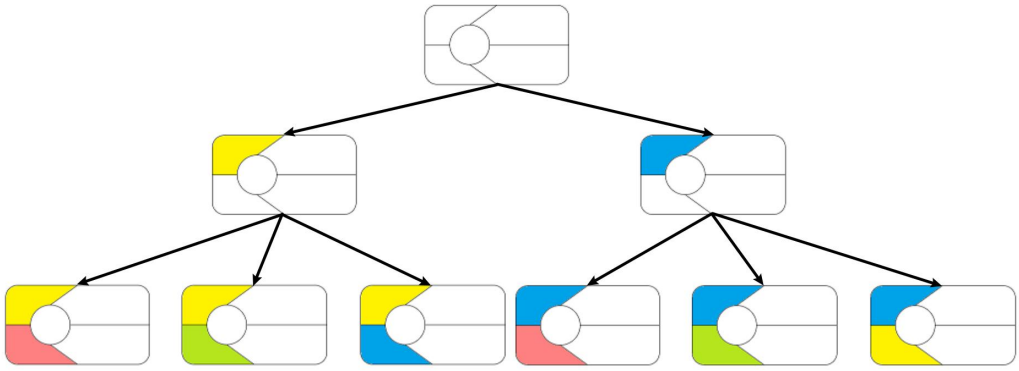


图 3 - 用于讲解“树层去重”的图

至此，我对我代码中的所有降低运行时间的方法都讲解完毕，在此做个总结。

1. 运用邻接表而不是邻接矩阵存储图的信息，这使得在寻找每个节点的相邻节点时不需要遍历整个邻接矩阵而是只需要遍历该节点的相邻节点数组即可。
2. 也即剪枝策略 1，向前探测——check 函数的实现
3. 也即剪枝策略 2，MRV+DH——选点策略的实现
4. 也即剪枝策略 3，“树层去重”的实现

❖ 根据实验要求，对附件中给定的地图数据填涂

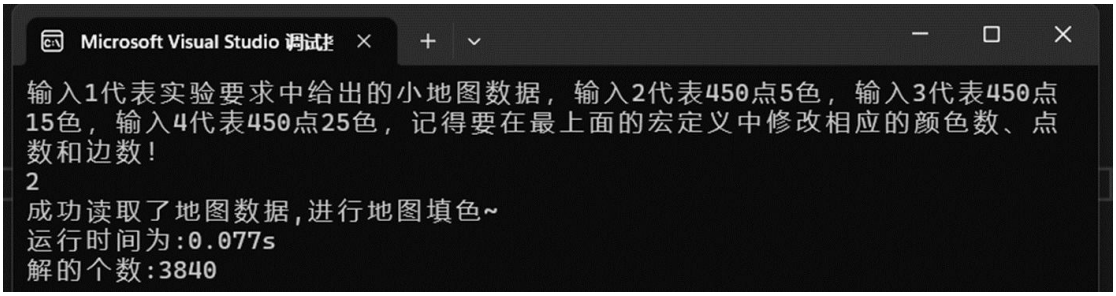
①得到第一个可行解的时间：

特别的，450 点 15 色我开始并不能得到第一个可行解的时间，在跟老师讲演此实验时老师告诉我可以将第一个要着色的点指定为编号为 4 的点，我在尝试后确实成功得到了第一个可行解的时间，感谢老师。

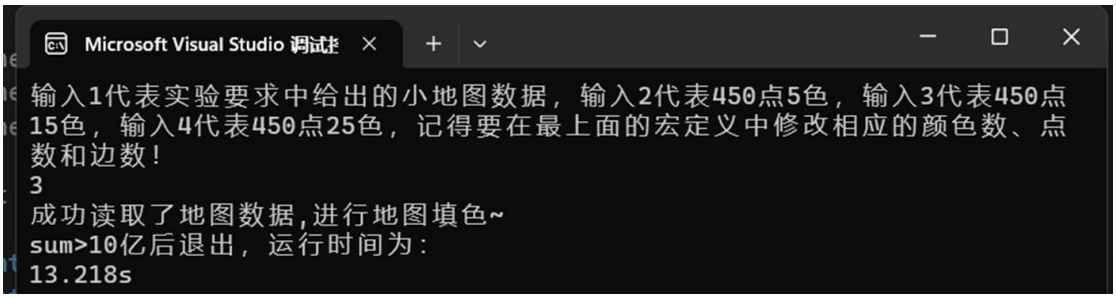
	450 点 5 色	450 点 15 色	450 点 25 色
运行时间	0.033s	0.147s	0.001s

②尝试得到全部可行解：

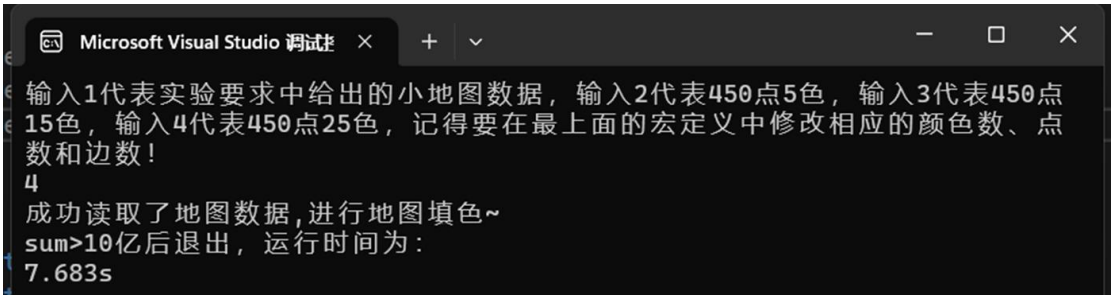
1. 对于 450 点 5 色图，可见经过重重优化后仅用 0.077s 就可得到全部的 3840 个解。



2. 对于 450 点 15 色图，我开始发现无法得到其全部解的个数，久久运行不出来。后我令解的数量 sum>10 亿后就退出来，得到 sum>10 亿后退出来的运行时间为 13.218s：



3. 对于 450 点 25 色图，我开始发现无法得到其全部解的个数，久久运行不出来。后我令解的数量 sum>10 亿后就退出来，得到 sum>10 亿后退出来的运行时间为 7.683s：



根据上面数据可以绘制下面表格：

	450 点 5 色	450 点 15 色	450 点 25 色
运行时间	0.077s	13.218s	7.683s
解的个数	3840	限制 sum>10 亿就退出	限制 sum>10 亿就退出

❖ 自行生成地图涂色并分析：

通过随机数生成器，我自行生成了不同的地图。我将程序运行 5 次而计算得到的运行时间平均值作为记录在表格和图像的运行时间。对于每次比较时，均需要保证其余变量的值不变，解的数量 sum>10 亿就退出，分析统计运行时间作表格和图像如下：

(1) 图规模对运行时间的影响：

注：对于不同的点数，边数均为点数的 2 倍，颜色总数均为 4。

	100 点 200 边	200 点 400 边	300 点 600 边	400 点 800 边
运行时间	13.049s	35.63s	51.339s	68.2158s

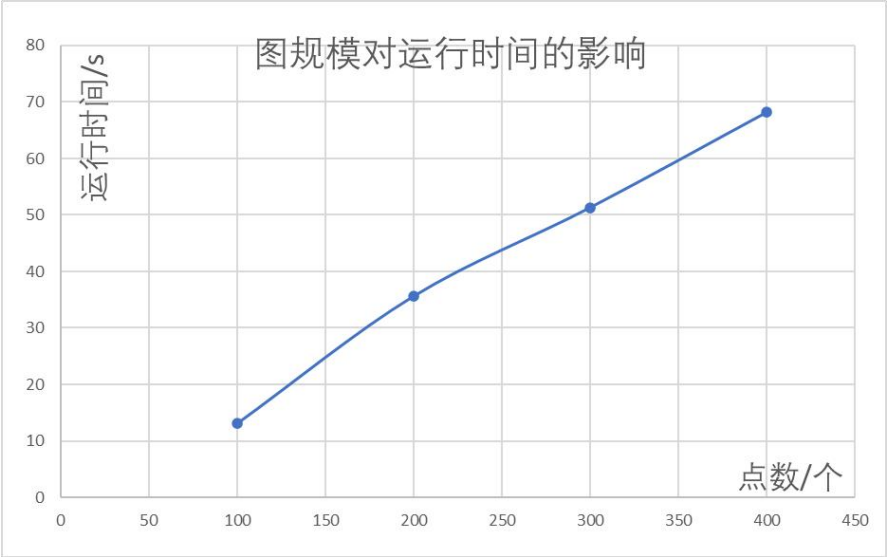


图 4 - 图规模对运行时间的影响

从上面的表格和曲线图可知：随着图规模的增大，算法的运行时间也随之增大。

分析原因：由于回溯树形结构同层节点即对应于图的同一个节点填不同颜色，故我们容易知道随着图规模的增大，点的个数增多，那么回溯树形结构的层数也就增多，整棵树也就更为庞大，故算法的运行时间也随之增大。

(2) 图的边数对运行时间的影响：

注：颜色总数均为 4。

	200 点 150 边	200 点 200 边	200 点 300 边	200 点 400 边
运行时间	48.9694s	42.1784s	38.0164s	32.444s

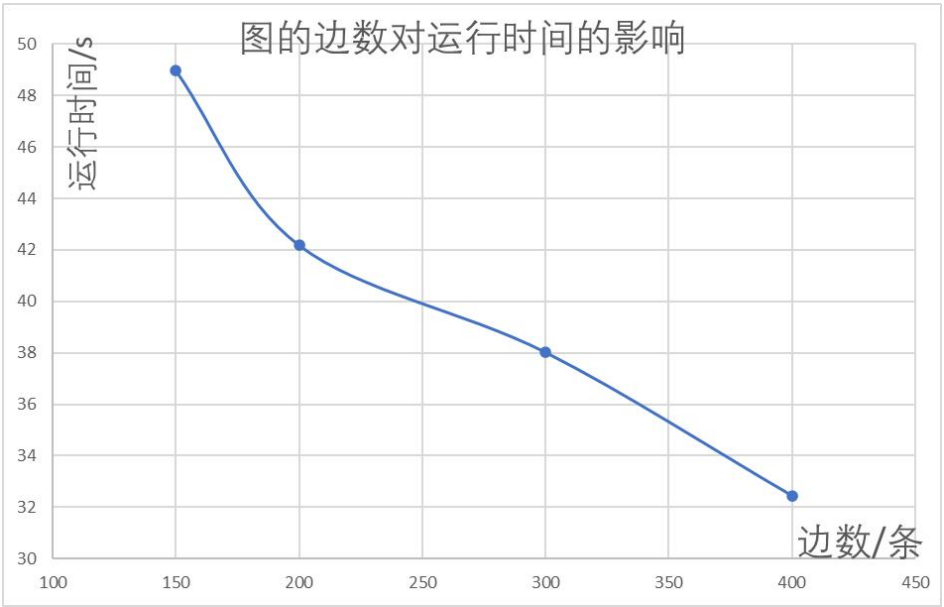


图 5 - 图的边数对运行时间的影响

从上面的表格和曲线图可知：在图的点数相同的情况下，图的边数越多，算法的运行时间就越小。

分析原因：由于图的边代表两个点之间的邻接关系，故在图的点数相同的情况下，图的边数越多就会导致点的平均度数增多（即点的平均相邻点个数增多），又因为在前面的向前探测剪枝时我们已知某个点填了颜色 c 后将会将会使其相邻节点都无法填颜色 c，即将会把其相邻节点填颜色 c 的分枝剪掉。现在随着边数的增加，点的平均相邻点个数增多了，那么对于某个点来说，它向前探测剪枝能剪掉的分枝就变多了。而点数不变意味着回溯树形结构的层数不变。在回溯树形结构层数不变的情况下能剪掉的分枝变多了，算法的运行时间自然就变小了。

四、实验结论和体会

结论：

这个实验让我学习了回溯法的基本思想和具体应用，让我感受到了回溯法的强大和灵活。我认识到了回溯法是是一种选优搜索法，又称为试探法，按选优条件向前搜索，以达到目标。我也知晓了如何用剪枝函数来优化回溯法，使其能够更快地找到问题的解。

体会：

a. 一开始直接想到的回溯算法往往效率较低，但我们可以通过仔细分析题目来思考出各种合适的剪枝策略来优化回溯算法，提高其效率。

b. 可以用数学逻辑思维或方法对回溯算法做出优化，例如在本题的剪枝策略 3 - “树层去重”中，我正是通过数学逻辑思维与方法对于同树层的很多节点进行了剪枝，从而大大缩小了算法的运行时间。

c. 除了考虑优化算法本身，选择合适的数据结构也能在一定程度上降低程序的运行时间。例如在本题中我选择使用邻接表而不是邻接矩阵来存储图的信息，这使得我得以更快的找到每个节点的相邻节点。

指导教师批阅意见： 成绩评定： 指导教师签字： 年　　月　　日
备注：

注：1、报告内的项目或内容设置，可根据实际情况加以调整和补充。

2、教师批改学生实验报告时间应在学生提交实验报告时间后 10 日内。