深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验项目名称：回溯法——地图填色问题

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：刘刚

报告人：林宪亮 学号：2022150130

实验时间：2024 年 4 月 27 日—2024 年 5 月 9 日

实验报告提交时间：2024 年 5 月 9 日

教务部制

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、实验内容：

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（黑白），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（Francis Guthrie）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（Wolfgang Haken）（生于1928年）和肯尼斯·阿佩尔（Kenneth Appel，1932年-2013年）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（Haken）成为伊利诺伊大学（University of Illinois）高级研究中心的成员，他现在是该大学的名誉教授。

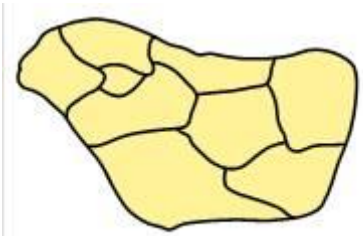
四色定理是第一个使用计算机证明的著名数学定理，此后变得越来越普遍，争议也越来越小 更快的计算机和更高效的算法意味着今天您可以在几个小时内在笔记本电脑上证明四种颜色定理。

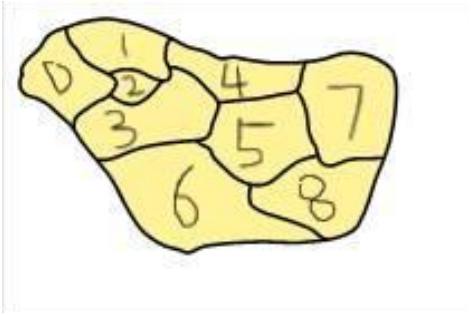
问题描述：

我们可以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，我们要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。

三、实验内容及过程：

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；





地图的邻接矩阵 G(如果 G[i][j]为 1 则说明 i,j 两个国家相邻,如果 G[i][j]为 0 则说明 i，j 两个国家不相邻)：

	0	1	2	3	4	5	6	7	8
0	0	1	1	1	0	0	0	0	0
1	1	0	1	1	1	0	0	0	0
2	1	1	0	1	0	0	0	0	0
3	1	1	1	0	1	1	1	0	0
4	0	1	0	1	0	1	0	1	0
5	0	0	0	1	1	0	1	1	1
6	0	0	0	1	0	1	0	0	1
7	0	0	0	0	1	1	0	0	1
8	0	0	0	0	0	1	1	1	0

思路：对于这个小规模的地图填色问题，我使用回溯法解决，回溯法的基本思想就是从问题的初始状态出发，一步一步地尝试所有可能的解决方案，并设置检查条件，当发现当前的解决方案已经不符合要求，则回溯到上一步，尝试另一种方案，直到找到所有符合要求的解决方案。而对于这个地图填色问题，则逐步尝试所有可以涂的颜色，并在涂完一个国家后，检查是否符合地图填色的要求，如果不符合条件则尝试另一种颜色，如果所有颜色都不符

合要求，则退回上一步，对上一个涂色的国家换一种颜色，检查后对下一个涂色的国家重复上面的步骤，如果当前涂色选择符合条件，则对下一个国家进行涂色，直到找到所有的可能解。

伪代码：

Algorithm 1：Map_Colored

Input：num_of_coloredcountries

Output: num_of_answers

1. If num_of_coloredcounties >= num_of_countries: num_of_answer++ then return num_of_answers
2. For i in range(num_of_colors):set the cur_countries color i, than check if each colored adjacent countries have different color;
3. If(have different color):Map_Colored(num_of_coloredcountries+1)
4. Else try the color i+1

说明：
num_of_coloredcountries 为当前已经涂色的国家数目，num_of_countries 为总共的国家数目，num_of_answer 为可行解的个数如果当前已经涂色的国家数目已经达到总共的国家数目，则让可行解的个数加一并返回。num_of_colors 为可选颜色个数。

运行程序，得出这个小规模地图有 480 个可行解，可验证回溯算法的正确性。

2、对附件中给定的地图数据填涂；

首先采用上诉未优化的回溯算法对给定地图进行涂色。发现即使是只找出一个可行解，所需要的时间已经超过了一小时。所以需要对我的回溯算法进行优化。

优化 1：地图存储结构的优化

首先我创建了一个 Vertex 类来表示地图中的国家，一个国家就是一个 Vertex，之后设置 Vertex 类的属性，包括它的颜色 color（初始化为 0 表示没有颜色），一个 able 数组用来记录它是否可以填涂对应的颜色（如 able[i]==1 则代表它可以填涂第 i 个颜色，这样避免了涂色之后再去遍历地图判断涂色是否正确，一个 num_of_choice 属性用来记录它还有多少种颜色可以选择，还有一个 du 属性记录这个点和多少个点相邻，用来寻找下一个涂色的点。

之后我改变了地图的存储方式，从原来的邻接矩阵改成了现在使用一个新的二维数组，数组的每一行表示一个国家，每一行的第一个元素表示它和多少个国家相邻，后面的每一个元素即是它相邻国家的编号。这样做减少了存储的规模，同时也降低了后续检查时遍历的时间开销。

优化 2：向前探测剪枝

即对当前节点涂色后，对当前节点相邻的节点进行遍历，如果因为当前节点的涂色后，导致存在相邻节点没有可以选择的颜色，那么说明当前涂色不合理，那么就不对这种涂色进行下一次探索，而是直接回溯对当前节点进行换色，这样就实现了剪枝操作。

例：

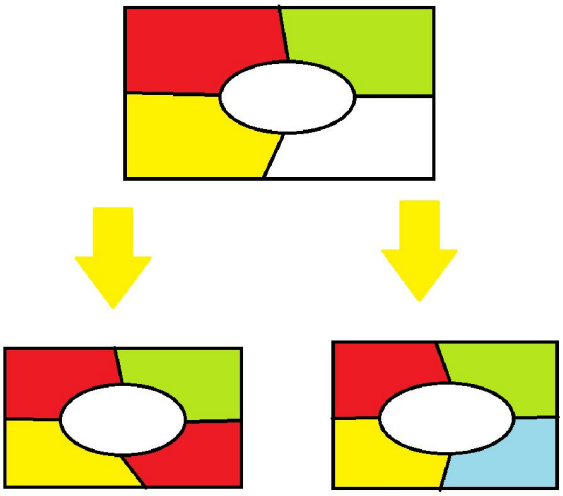


图 1 剪枝

如图，如果我们有红蓝黄绿四种颜色可以使用，那么当我们如左侧进行填涂时，则还有可能性填涂完这张地图，那么则不需要回溯，如果我们如右侧进行填涂，那么中间的圆形区域已经没有颜色可以进行填涂，所以这张情况就无需再进行向下探索，直接回溯即可。

伪代码：

Algorithm 2：Map_CUT

Input：VertexArr，cur_index，color_i

Output: True/False

```
1. For every adjacent vertex：
2.         If the vertex.color==0 and vertex.able[i]=1:
3.                 Vertex.able[i]=-cur then vertex.choice--
4.                 If vertex.choice == 0    return False
5. Return True
```

说明：对于当前节点的所有相邻节点，如果它没有涂色并且可以涂当前节点所涂的颜色，那么设置该相邻节点的该颜色的涂色状态为-cur（表示不可涂，设置为-cur 是为了方便后续回溯），然后对该相邻节点的选择个数减一，之后判断该相邻节点是否还有可以选择的填涂颜色，如果没有，那么就不用再继续向下探索了，进行回溯。

优化 3：下一个填涂节点的选择策略

如果当前节点填涂后符合要求，那么就需要对下一个节点进行填涂，为了降低程序所需要的时间，科学的选择下一个填涂的节点十分的重要。对于下一个填涂节点的选择，我采取以下两个策略：

（1）在待涂色的点中选则可选颜色最少的点。

（2）在可选颜色最小的节点中选择度最大的点。

选择可选颜色最少的点增大了失败的概率，所以一定程度上提前了回溯操作，这样实现了剪枝。

选择度最大的点可以增大当前涂色影响的点的个数，这样也可以更快的发现不符合条件的涂色方法，提前回溯，实现了剪枝操作。

伪代码：

Algorithm 3：Map_NEXTVERTEX

Input：VertexArr

Output: NextVertex

```
1. Set Min=num_of_colors,Next=0
2. For every Vertex:
3.     If the vertex is not colored:
4.         If vertex.choice=Min and vertex.du > Next.du:
5.                 Next=vertex
6.         If vertex.choice < Min:
7.                 Min=vertex.choice
8.                 Next=vertex
9. Return Next
```

说明：

初始化可选择的最少颜色为最大值（所有可选的颜色数），下一个节点默认为第 0 个节点。之后遍历每一个节点，如果这个节点没有被涂色并且它的可选颜色少于 Min，那么这个点就是下一个涂色的点，如果这个点的可选颜色等于 Min 但是它的度数更大，那么设定它为下一个涂色的点，这是优先考虑可选颜色次考虑度的选择策略，也可以尝试优先考虑度次考虑可选颜色的策略。但经过实验证明优先考虑可选颜色次考虑度的选择策略的确是更优的策略。

优化 4：对称性工作的去重

思考：如果一个节点涂了一个新的颜色，如果在涂了这个颜色后，这个地图拥有 n 个可行解，那么这个节点填涂别的新的颜色后，也会拥有 n 个可行解，这个关键就在于新的颜色，都是全新的颜色，那么它们理应拥有相同个数的解。这样就可以实现去重，降低开销。

例子：

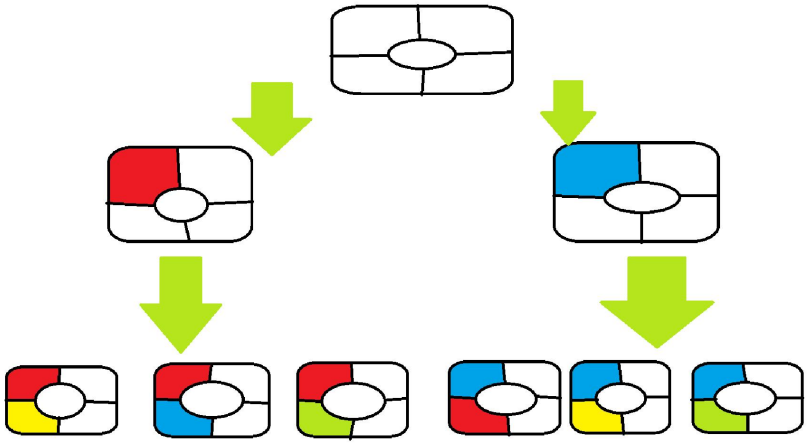


图 2 对称去重

如图，对于填涂一个新的颜色（红色或者蓝色），它们后续对应的可行解个数应该是一样的，这具有一种对称性。

经过四步优化后的伪代码：

Algorithm 4：Map_Colored_Improved

Input：VertexArr,curindex,count,num_of_used_colors

Output: num_of_ans

```
1. If count==Num_of_vertex: num_of_ans +=curVertex.choice then return
2. Else for i in range(Num_of_Colors):
3.       If(the color[i] is useful for the cur vertex):
4.           If(Map_CUT()):
5.               If(color[i] is a new color):
6.                   Num1=Map_Colored_Improved(VertexArr,
Map_NEXTVERTEX(),count++,num_of_used_colors+1)
7.               Else
8.                   Num1=Map_Colored_Improved(VertexArr,
Map_NEXTVERTEX(),count++,num_of_used_colors)
9.           Set curVertex.color=0 than recover the data changed because the curVertex
10.          If color[i] is a new color :
11.              Num_of_ans+=Num1*(num_of_colors-num_of_used_colors-1) then break
12. Return
```

说明：这是经过四次优化后的地图填色算法代码。对于算法的当前状态，先判断是否已经所有点都涂色完毕，如果是则变更 num_of_ans 即更新可行解个数并返回。如果没有全部涂色完毕则进入另一个分支，在另一个分支中尝试对当前节点进行涂色，涂色后会使用 Map_Cut()算法进行判断涂色是否满足条件，不满足则会进行剪枝回溯。如果通过了检查，那么就会进一步判断当

前涂色的颜色是否为一个新的颜色，之后递归调用函数本身进行求解。如果当前节点所涂的颜色是一个全新的颜色，那么就可以不用对剩下的颜色进行尝试了，直接在 num_of_ans 上加上一个当前涂色可行解的一个倍数即可。

对提供的地图数据进行涂色：

表一： 地图涂色数据（第一个可行解）

N（ms） 地图 类型	1	2	3	4	5	6	平均
5 色 450 点	48	49	53	51	47	46	49
15 色 450 点	162	162	163	161	167	161	162.67
25 色 450 点	0	1	1	0	1	0	0.5

如上表，在 5 色 450 点的地图中，要找到一个可行解所需要的时间大概为 49ms，在 15 色 450 点的地图中，要找到第一个可行解的时间大概为 162.67ms，在 25 色 450 点的地图中，要找到第一个可行解的时间只需要 0.5ms，是最快的。

表二： 地图涂色数据（所有可行解）

N（ms） 地图 类型	1	2	3	4	5	6	平均
5 色 450 点	118	117	121	113	112	110	115.16
15 色 450 点	∞	∞	∞	∞	∞	∞	∞
25 色 450 点	∞	∞	∞	∞	∞	∞	∞

如上表，在 5 色 450 点的地图中，找到所有可行解的平均时间为 115.16ms，共有可行解 3840 个。对于剩下的两个地图，可行解的个数实在是太多了（超过了 100 亿个），要找出所有可行解的时间开销也是太大，因此记录为正无穷。

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）。

• 首先，我随机选取一些规模的地图数据进行测试，大致掌握一定规模的地图数据所需要的时间，方便后续的实验测试进行。

表三：前期测试数据

N（us） 地图 类型	1	2	3	4	5	平均
100 点 100 边	0	0	0	0	0	0
200 点 200 边	0	0	0	0	0	0
300 点 300 边	0	0	0	0	0	0
400 点 400 边	1000	0	1000	0	1000	666
500 点 500 边	1000	1000	1000	1000	1000	1000

• 对一定规模数据所需要的时间有一点了解后，我开始使用随机地图数据开始实验，下表测试的是在边数固定的情况下，点数的增大和算法求出第一个可行解的时间的关系。

表四：点数和计算第一个可行解时间的关系

N（us） 地图 类型	1	2	3	4	5	平均
100 点 1000 边	0	0	0	0	0	0
200 点 1000 边	0	0	0	0	0	0
300 点 1000 边	1500	1600	1400	1700	1300	1500
400 点 1000 边	2100	2200	1800	1900	2000	2000
500 点 1000 边	3000	3200	2800	3000	3000	3000
600 点 1000 边	3700	3900	3800	3800	3800	3800
700 点 1000 边	5000	5000	5000	5000	5000	5000
800 点 1000 边	7400	7700	7500	7400	7500	7500
900 点 1000 边	8500	9100	8800	8700	8700	8760
1000 点 1000 边	11000	13000	11000	15000	10000	12000

由上表的结果可以知道，边数固定的情况下，随着点数的增多，求解出第一个可行解的时间也会增大，因为点数越多，消耗的资源也就越多，解的搜索空间更大，搜索时间更长。

•在求解 10 亿个可行解所需要的时间和顶点数的关系中，同样也满足这个规律，如下表：

表五：10 亿个可行解所需要的时间和顶点数的关系

规模	200 点 1000 边	300 点 1000 边	400 点 1000 边	500 点 1000 边
时间（s）	24.41	29.32	37.53	41.45

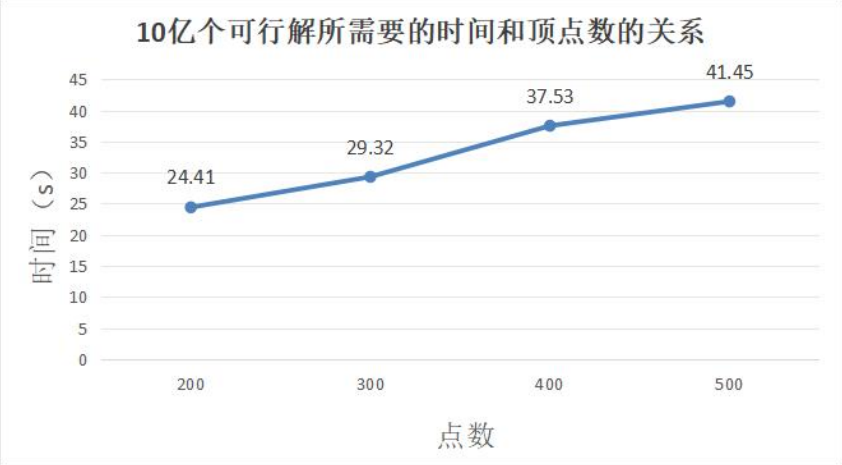


图 3 10 亿个可行解所需要的时间和顶点数的关系

•下面探索在点数不变的情况下，边数增多对算法求出第一个可行解的时间的影响。

表六：边数与第一个可行解求解时间的关系（固定点数为 500）

边数	200	400	600	800	1000	1200	1400
时间（us）	0	1000	1400	1900	3000	2200	1500

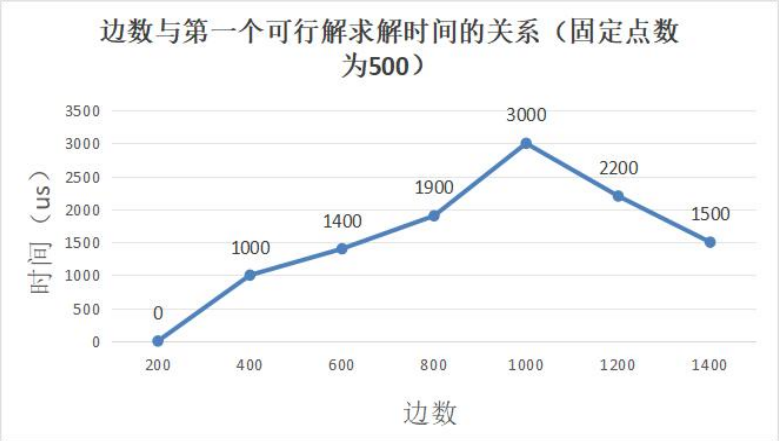


图 4 边数与第一个可行解求解时间的关系（固定点数为 500）

由上表的结果可知，算法所需要的时间先随着边数的增加而增加，这是因为搜索空间增加了，但是当边数达到一定数量级后，算法所需的时间反而会减少，这是因为变密度增大，图变得复杂，那么回溯的效率就会更高，使得搜索效率更高。

• 下面探索在点固定的情况下，随着边数的增加，求解出 10 亿个解的所需时间是否也符合这个规律。

表七：边数与求解 10 亿个可行解所需时间的关系（500 点）

边数	300	600	900	1200	1500	1800
时间（s）	41.32	54.45	61.34	45.51	36.31	29.41

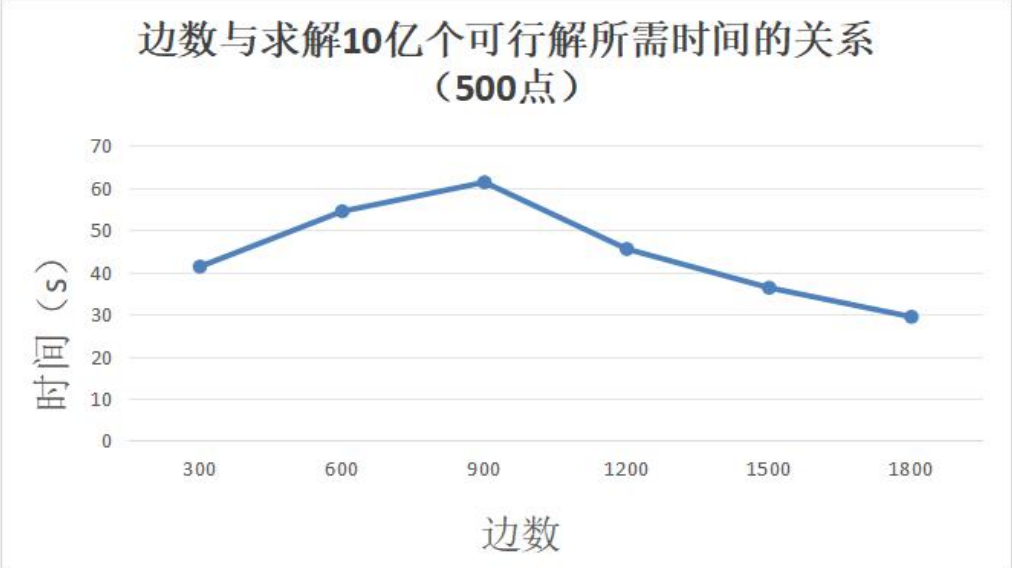


图 5 边数与求解 10 亿个可行解所需时间的关系（500 点）

由上表结果可知，在求解 10 亿个可行解时，算法所需要的时间也和边数的增加呈现一种先增大后减小的关系，也符合这个规律。

四、实验结果及分析：

本次实验，我先通过简单基础的回溯算法对小规模地图进行填色，得出共有480种涂色方案，验证了算法的正确性。但是使用基础的回溯算法对实验提供的地图数据进行填色时，发现运行了一个多小时也跑不出一个可行解，于是对于现有的算法进行了四次优化，分别是（1）地图存储结构的优化（2）向前探测剪枝优化（3）下一个填涂节点的选择策略优化（4）对称性工作的去重，四步优化的具体细节在上述实验报告中都有给出，经过优化后，算法可以成功的找出5色450点地图的所有3840个解，并且只需要十秒钟左右，也可以在较短时间内找出15色和25色地图的可行解，但是依旧无法在一个小时内找出所有的可行解，于是我对代码进行了修改，如果找到的可行解个数超过100亿个，那么程序就不再进行运行，而是退出，修改后发现15色和25色的地图的可行解都超过了100亿个，推测是可行解的个数实在是太多了，所以无法在可以接受的时间内找出所有的可行解。

之后，我又使用随机产生的地图数据进行实验，发现了一些规律（1）随着地图点数规模的增大，无论是找到第一个可行解还是找到10亿个可行解的时间都在增大，推测是因为边数固定的情况下，随着点数的增多，求解出第一个可行解的时间也会增大，因为点数越多，消耗的资源也就越多，解的搜索空间更大，搜索时间更长。（2）随着地图边数规模的增大，找到第一个可行解和10亿个可行解的时间都呈现先上升后下降的趋势，推测是因为搜索空间增加了，但是当边数达到一定数量级后，算法所需的时间反而会减少，这是因为变密度增大，图变得复杂，那么回溯的效率就会更高，使得搜索效率更高。

五、实验结论：

1. 使用未优化的回溯算法无法在可接受的时间内找出即使一个可行解。
2. 对于地图涂色的优化方式有四个（1）地图存储结构的优化（2）向前探测剪枝优化（3）下一个填涂节点的选择策略优化（4）对称性工作的去重
3. 经过实验证明，在下一个填涂节点的选择策略上，优先颜色次度的选择策略会更加高效。
4. 实验提供的15色和25色地图的可行解都超过了100亿个，难以在可接受的时间内找出所有的可行解。
5. 随着地图点数规模的增大，无论是找到第一个可行解还是找到10亿个可行解的时间都在增大。
6. 随着地图边数规模的增大，找到第一个可行解和10亿个可行解的时间都呈现先上升后下降的趋势。

指导教师批阅意见：   成绩评定：   指导教师签字： 2024 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。