深 圳 大 学 实 验 报 告

课程名称：算法设计与分析

实验名称：实验 5——图论桥问题

学院：计算机与软件学院 专业：软件工程

报告人：郑杨 学号：2020151002 班级：腾班

同组人：陈敏涵

指导教师：李炎然

实验时间：2022.6.17~2022.6.20

实验报告提交时间：2022.6.20

教务处制

目录

一、实验目的

1、掌握图的连通性。
2、掌握并查集的基本原理和应用。

二、实验内容与要求

1. 桥的定义

在图论中，一条边被称为“桥”代表这条边一旦被删除，这张图的连通块数量会增加。等价地说，一条边是一座桥当且仅当这条边不在任何环上。一张图可以有零或多座桥。

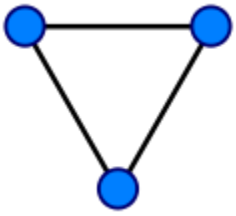


图 1：没有桥的无向连通图

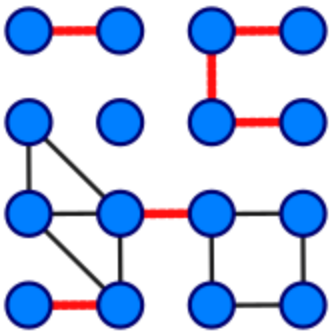


图 2：这是有 16 个顶点和 6 个桥的图（桥以红色线段标示）

2. 求解问题

找出一个无向图中所有的桥。

3. 算法

（1）基准算法

For every edge (u, v), do following

a) Remove (u, v) from graph

b) See if the graph remains connected (We can either use BFS or DFS)

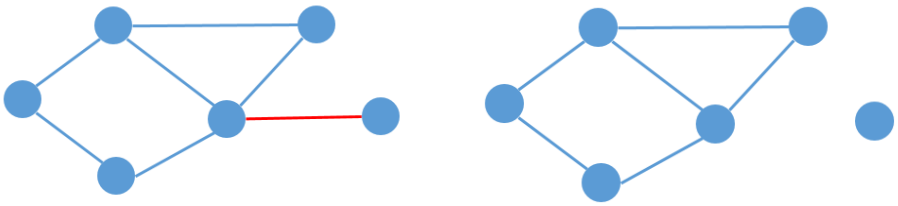
c) Add (u, v) back to the graph.

（2）应用并查集设计一个比基准算法更高效的算法。不要使用 Tarjan 算法，如果使用 Tarjan 算法，仍然需要利用并查集设计一个比基准算法更高效的算法。

三、实验步骤与结果

1 问题分析

首先对这次实验的问题进行分析，然后提出若干解决方案对该问题进行求解。该问题描述起来比较简单，给定一个无向图$G=(V,E)$（不一定连通），要求找出所有的“桥”。“桥”的定义就是，一条边$e\in E$如果从图G中删除掉之后，图中的连通分量个数增多，则该边为图中的一座“桥”。如图 3(a)所示，一开始图中连通分量的个数为 1，去掉红色的边之后，如图 3(b)所示，连通分量个数变为 2，故红色边是一座“桥”。从定义出发，可以得到下一节将叙述的基准算法。



(a) (b)

图 3：问题描述

2 基准算法

2.1 基准法 1

基准算法 1 从定义出发，也算是一种暴力求解方法。

2.1.1 算法思想

从第 1 节问题分析中可以清楚的知道桥的定义，不难想到一种暴力的求解方法。首先求出图G中的连通分量个数（记为p），然后枚举图中每一条边，对于每一条边e，把它从图

G 中删除，求出删除之后的图 $G'(V, E-\{e\})$ 的连通分量个数 p'，若 $p' > p$，则说明边 e 是图 G 的一座“桥”。

那么问题转化为，求图 $G(V, E)$ 中的连通分量个数。可以通过 dfs 或者 bfs 的搜索算法进行求解，大概思路是，使用 dfs 或者 bfs 搜索从每一个为搜索过顶点开始对图 G 进行搜索，每一次都会搜索一个连通子图，那么只需要统计连通子图的次数就可以了，具体流程见 2.1.3 小节的算法流程。

2.1.2 算法流程

以图 4 所示的简单例子叙述基准算法 1 的算法流程，原始的图 $G(V, E)$ 如图 4 所示，包含 16 个顶点和 15 条边。首先以 dfs 为例遍历整张图，计算图 G 中的连通分量个数。

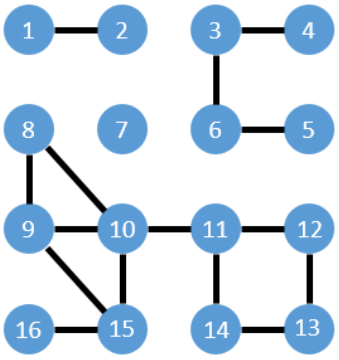


图 4：简单例子

如图 5 所示，首先以顶点 1 作为 dfs 的起点，往其邻接点进行搜索，知道无法往下搜索为止。可以看到，顶点 1、2 在第一次搜索中都被搜索到，故它们同为图 G 中的一个连通分量，把图 G 的连通分量个数加 1。在算法进行到顶点 2 时，由于顶点 2 已被搜索过，故跳过该顶点继续前往下一个顶点重新开始对图进行搜索。

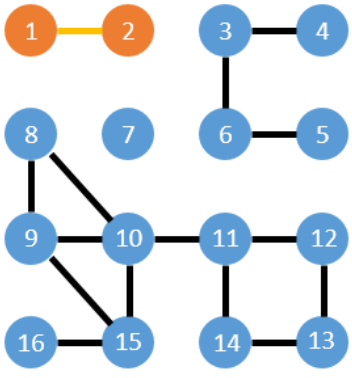


图 5：基准法 1 算法流程（1）

在整个 dfs 流程结束之后，图中的连通分量分布如图 6 所示，可以看到，在枚举每一条边找桥之前，图中的连通分量个数为 4。

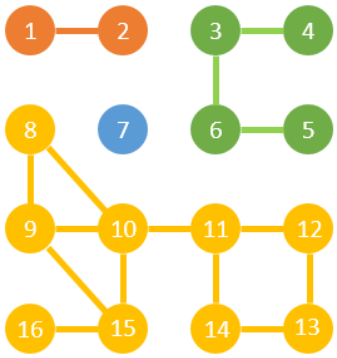


图 6：基准法 1 算法流程（2）

在统计完图G中的连通分量p之后，开始枚举每一条边判断是否该边为桥，首先枚举连接顶点 1 和顶点 2 的边，将该边删除，求出此时图G'的连通分量个数p'。如图 7 所示，该边删除之后图G'连通分量个数p'增加了，故该边为图G中的一座桥。

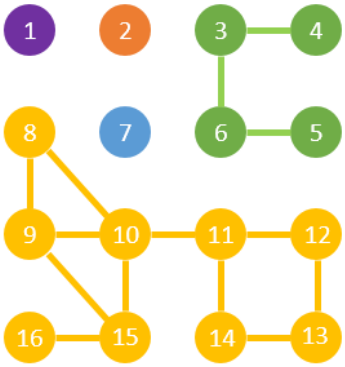


图 7：基准法 1 算法流程（3）

在删除边并判断该边是否为桥之后，还需要把该边补回去，然后继续枚举下一条边进行判断，使用类似的流程即可，最终可以求出图G中所有的桥，如图 8 所示。

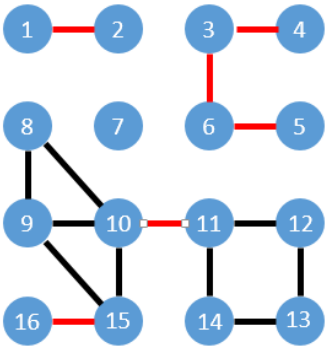


图 8：基准法 1 算法流程（4）

2.1.3 算法复杂度分析

假设图G的点数为n，边数为m。在进行桥的判定前，需要先算出图G的连通分量个数，算连通分量的过程相当于在遍历图G的n个点，故需要$O(n+m)$的时间；在枚举每一

条边进行桥的判定时，对于每一条边都需要计算一遍删去该边之后图G'的连通分量个数，于是需要$O(m(n+m))$的时间。于是，基准法 1 的时间复杂度为：$O(mn+m^2)$。

基准法 1 的空间复杂度取决于存图的数据结构的选择，这里我选择使用邻接表进行图结构的保存，于是基准法 1 的空间复杂度为$O(m)$。

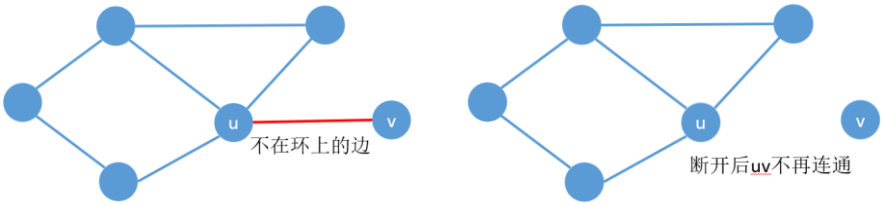
2.1.4 算法效率测试

算法效率测试统一在 5.1 节中呈现。

2.2 基准法 2

2.2.1 算法思想

由桥的定义：一条边是一座桥当且仅当这条边不在任何环上，可以通过与基准法 1 不同的方式进行桥的判断。如果一条边是桥的话，那么它必然不在任何环上，那也就是说，这条边一旦断开，边的两个顶点将不再连通，如图 9 所示；而如果它在一个环上，那么它必然不是一座“桥”，此时就算断开这条边，它的两个顶点仍存在一条路径使得它们连通，如图 10 所示。



(a)桥边断开前　　(b)桥边断开后

图 9：基准算法 2 思想（1）

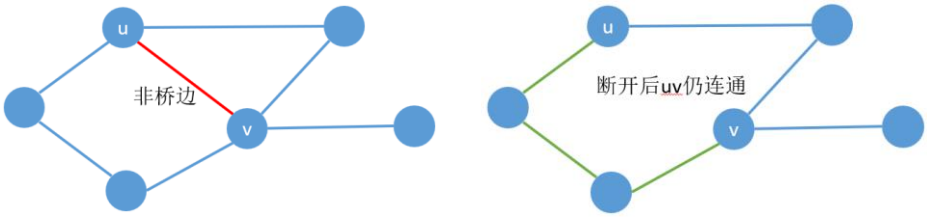


图 10：基准算法 2 思想（2）

那么依据桥边与非桥边这个不一样的特性（断开后边的两顶点是否连通），仍然枚举每一条边，将该边断开，判断是否仍存在一条路径，从边的一个顶点到达另一个顶点，如果不存在则该边为“桥”，否则反之。寻找路径可以使用 dfs 或者 bfs，判断从一个顶点出发的 dfs 树或者 bfs 树是否存在另一个顶点。

2.2.2 算法流程

仍以图 4 所示的简单例子叙述基准算法 2 的算法流程，与基准算法 1 类似，首先枚举边进行删除，如图 11 中所示，此时删除的边为连接顶点 1 与顶点 2 的边，由于删除之后从顶

点 1 出发无法到达顶点 2，故该边为一条桥边。

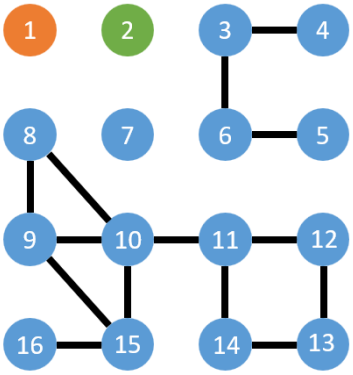


图 11：基准算法 2 算法流程（1）

在这之后，把删除的边添加回图中，枚举剩下的边进行判断，当枚举到连接顶点 8 与顶点 9 的边时，如图 12 所示，由于在该边删除之后，顶点 8 与顶点 9 仍存在一条路径（8 → 10 → 9）将它们连通，故该边不是桥边。

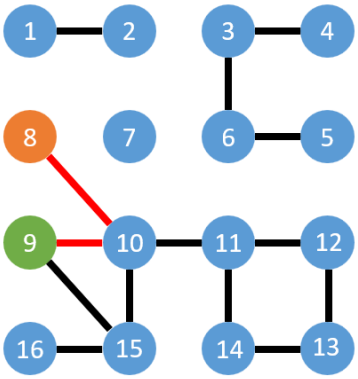


图 12：基准算法 2 算法流程（2）

按照类似的方法完成剩下的边的枚举，即可得到与基准算法 1 一致的结果。

2.2.3 算法复杂度分析

假设图G的点数为n，边数为m。基准算法 2 在进行桥的判定时，需要枚举m条边，对于每一条判定中的边，需要从边的一个顶点搜索到边的另一个顶点，平均的搜索代价为$\frac{n+m}{2}$，故基准算法 2 的时间复杂度为$O\left(m\frac{n+m}{2}\right)=O(m^2+nm)$。与基准算法 1 相比，理论上常数偏小。

空间复杂度取决于存图的数据结构的选择，这里我选择使用邻接表进行图结构的保存，于是基准法 1 的空间复杂度为$O(m)$。

2.2.4 算法效率测试

算法效率测试统一在 5.1 节中呈现。

3 优化基准法

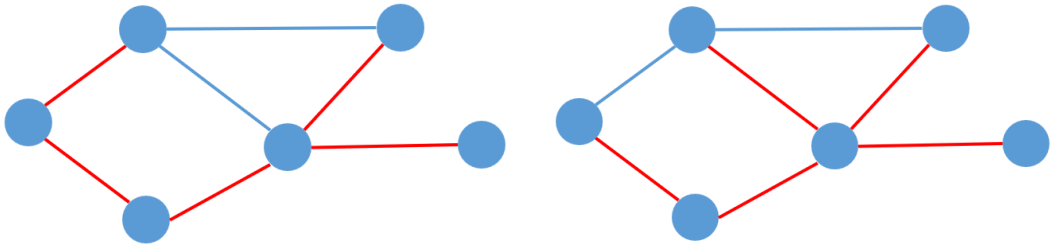
基准法 1 与基准法 2 的时间复杂度都与边数点数有关，理论上在稠密图的数据上运行效率极为低下，可以进一步进行优化。实际上，基准法 2 已经应用了图论中桥的一个性质，就是桥边必然不在图中的任意一个环上，也就是说桥边是图中连接该边的两个顶点的唯一路径。

3.1 生成树与生成森林

无向连通图 $G(V,E)$ 的生成树指的就是一颗满足以下条件的树 $T(V_t,E_t)$：

$$V_t=V,E_t\subset E,|E_t|=|V|-1$$

如图 13(a)所示，图中标示了一颗生成树。生成树不唯一，如图 13(b)所示，标示了图 13 所示的图中的另一颗生成树。



(a) (b)

图 13：无向图的生成树（1）

由于生成树需要保证原图的所有顶点连通，不难得到一个结论，生成树必然包含原图所有的桥边。可以使用反证法证明这个结论：

假设生成树不包含原图中的某一条桥边 (u,v)，那么由桥的性质得到，将该桥边去掉之后，该边的两个顶点 u,v 不存在另一条路径使得它们连通，这与生成树的定义相矛盾，故生成树必然包含原图所有的桥边。

然而，并不是所有的生成树边都为桥边，如图 14 所示，绿边虽然为生成树边，但并不是原图的桥边。

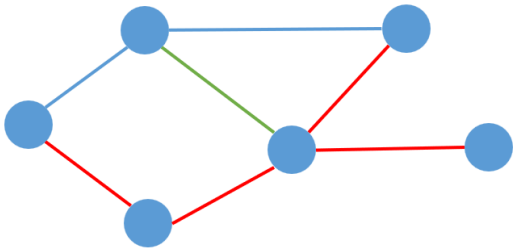


图 14：无向图的生成树（2）

推广到无向图（不一定连通）中，由于图不一定连通，故可能存在多颗生成树，如图 15 所示，称为生成森林，森林中的每一颗生成树对于上述结论都成立。

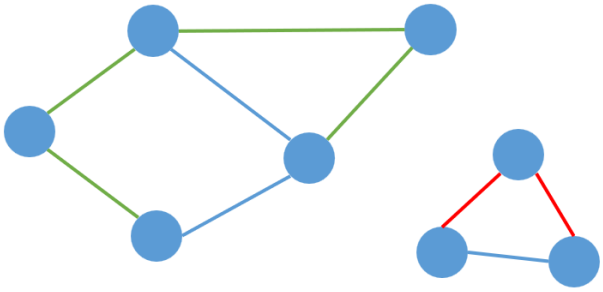


图 15：无向图的生成森林

3.2 生成树/生成森林的生成

在理清楚生成树与生成森林的概念之后，需要对它们进行生成。由于 dfs 或 bfs 遍历图结点的时候，本质上就已经生成了一颗生成树（或者生成森林），于是只需要在 dfs 或者 bfs 遍历图结点的时候顺便记录下生成树的边即可。仍以图 2 的简单例子为例，如图 16 所示，从每个未标记顶点开始搜索一颗生成树，最后生成了一片生成森林（以不同颜色区分不同的生成树）。

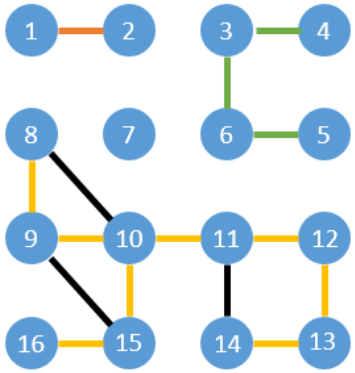


图 16：生成森林的生成

3.3 优化算法思路

在上述生成树与生成森林的介绍中得出了一个结论：无向图的所有桥边都被包含在该图的生成树或者生成森林中。于是在基准法 1 或者基准法 2 中，枚举图中所有边进行判断是没有必要的，只需要枚举该图的任意一颗生成树或者任意一个生成森林中的边进行判断即可。具体流程在下一小节进行阐述。

3.4 算法流程

首先使用生成树的构造算法构造原图的一颗生成树（生成森林），然后枚举生成树（生成森林）上的所有边，使用基准法 1 或者基准法 2 中的桥判定算法进行是否为桥的判断。具体判断过程与基准法 1 或基准法 2 中的判断过程一致，故不再赘述。

3.5 算法复杂度分析

由于空间复杂度不变，这里只讨论优化后的时间复杂度。

3.5.1 基准法 1 优化后的时间复杂度

假设图G的点数为n，边数为m。在进行桥的判定前，需要先构造出图G的生成树（生成森林），构造生成树（生成森林）的过程相当于在遍历图G的n个点，在构造的同时可以顺便计算出图G的连通块个数（即森林中的生成树数量），故需要$O(n+m)$的时间；在枚举生成树（生成森林）的每一条边即$n-1$条边进行桥的判定时，对于每一条边都需要计算一遍删去该边之后图G'的连通分量个数，于是需要$O\big((n-1)n\big)=O(n^2+nm)$的时间。于是，基准法 1 优化后的时间复杂度为：$O(n+m)+O(n^2+nm)=O(n^2+nm)$。

3.5.2 基准法 2 优化后的时间复杂度

假设图G的点数为n，边数为m。在进行桥的判定前，需要先构造出图G的生成树（生成森林），构造生成树（生成森林）的过程相当于在遍历图G的n个点，故需要$O(n)$的时间；在进行桥的判定时，需要枚举$n-1$条边，对于每一条判定中的边，需要从边的一个顶点搜索到边的另一个顶点，平均的搜索代价为$\frac{n}{2}$，故基准算法 2 优化后的时间复杂度为$O\left(n+(n-1)\frac{n}{2}\right)=O(n^2)$。与基准算法 1 相比，理论上常数偏小。

3.5.3 优化后的时间复杂度总结

可以看到，优化后的时间复杂度只与图中的点数有关，理论上在稠密图中表现比优化前的基准法高效，而在稀疏图上则差不多。这在第 5 节中会进行相关测试说明。

4 高效算法

仍然以生成树（生成森林）作为出发点进行优化，上述所有算法都是针对于桥边的判断，接下来介绍的高效算法基本上都是从环边出发，也就是排除掉生成树（生成森林）中的边中的环边，剩下的就都是桥边了。

4.1 并查集+最近公共祖先（LCA）求桥

4.1.1 基本算法思想

基于排除生成树（生成森林）中环边的思想，构思出这样的一个算法。由于生成树中每多一条边，就会形成一个环，如图 17 所示，红边会使得生成树中的顶点 1、2、4 构成一个

环，边$(1,2),(2,4)$与红边构成环，于是成为环边。

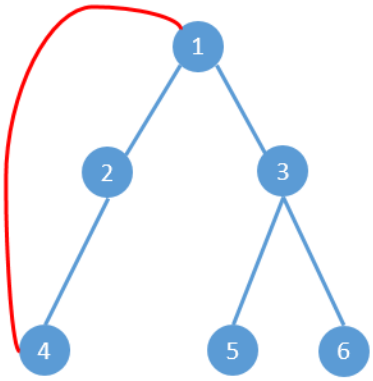


图 17：生成树加一边构成环

于是，在构造出原图的一颗生成树（生成森林）之后，可以通过枚举所有的非树边（不在生成树上的边），把形成的环上的树边（在生成树上的边）标记为环边。最后排除所有树上的环边就可以得到剩下的桥边了。

如何找到与枚举的那一条非树边与树边形成的环呢？可以看到，如图 17 所示，非树边$(1,4)$的两个顶点在树上的路径$1 \to 2 \to 4$与该非树边构成了环。那么问题转化为，找到非树边的两个顶点在树上的路径，并把该路径上的所有边标记为环边。在介绍此问题的解决方案之前，先引入一个概念：树上最近公共祖先。

4.1.2 最近公共祖先

最近公共祖先简称 LCA（Lowest Common Ancestor），两个节点的最近公共祖先，就是这两个点的公共祖先里面，离根最远的那个。如图 18 所示，节点 5 和节点 6 的公共祖先有 1 和 2，其中离跟最远的公共祖先是 2，故$LCA(5,6) = 1$。

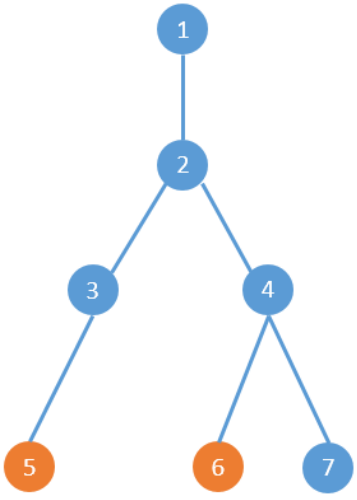


图 18：两节点的最近公共祖先

由上图可以清晰的看出，树上任意两个节点a, b的路径为$a \to LCA(a,b) \to b$，如图 19 中的节点 5 和节点 6，它们在树上的路径如图中所示，为$5 \to 3 \to LCA(5,6) = 2 \to 4 \to 6$。故，树上的任意两个节点的路径可以通过求解它们的树上 LCA 间接得到。这也就是 4.1.1 节中提出问题的一个解决方案，而这么做是为了之后更好地优化。

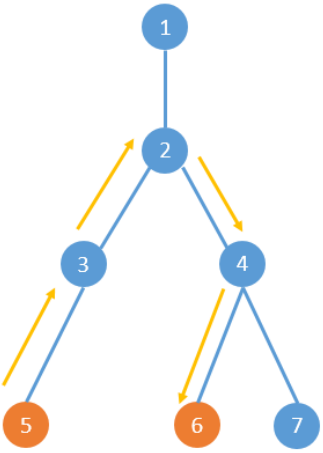
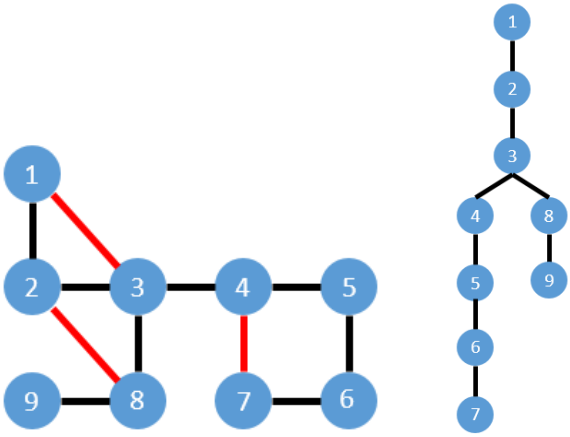


图 19：两节点树上路径

4.1.3 基本算法流程

按照上面的思路，首先需要构造原图的生成树（生成森林），然后枚举所有的非树边，找到该边两个顶点在生成树上的路径（通过找到 LCA），然后对路径上的边进行标记（可以在找 LCA 时进行标记）。最后，不带标记的边就为桥边。

以图 20(a)的例子为例观察基本算法的流程，首先构造如图 20(b)的一颗生成树，非树边在原图中以红边标记。



(a) (b)

图 20：例子

首先考虑非树边(1,3)，如图 21 所示，求解节点 1 和节点 3 的 LCA 时，将深度较大的节点 3 往上走，经过的边分别为(3,2)(2,1)，最终 LCA 为节点 1，被标记为环边的边有(3,2)和(2,1)。

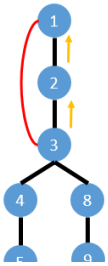


图 21：算法流程（1）

之后考虑非树边(2,8)，如图 22 所示，求解节点 2 和节点 8 的 LCA 时，将深度较大的节点 8 往上走，经过的边分别为(8,3)(3,2)，最终 LCA 为节点 2，被标记为环边的边有(8,3)和(3,2)。

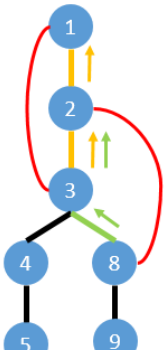


图 22：算法流程（2）

之后考虑非树边(4,7)，也是类似的操作，最后查看树边中所有未标记的树边，即为桥边。如图 23 所示，桥边为(3,4)和(8,9)。

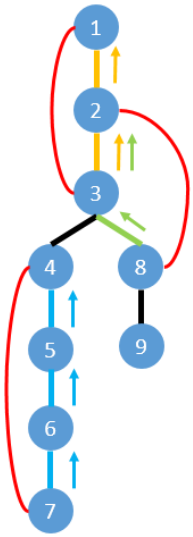


图 23：算法流程（3）

4.1.4 基本算法存在的问题

观察 4.1.3 中的流程可以看出，树边(3,2)在算法运行过程中会被标记两遍，这属于一个冗余操作。在图的规模增大时，这种冗余操作会越来越多，极大了降低了算法的运行效率。可以看到，这种情况是两个环有交集的时候才会产生的，可以在标记环边的过程中把环内的点都压缩起来，在之后其他非树边产生的环与已有的环发生交叠时，就不需要再标记已有的环内的边了（因为在之前已经标记过了）。如图 24 所示，在标记非树边(1,3)产生的环时，把节点 1、2、3 压缩成一个大点，当在标记非树边(2,8)产生的环时，就不会重复标记树边(3,2)了。

压缩点的操作可以通过一种数据结构——并查集进行维护，接下来先介绍这种数据结构，再进一步对优化后的算法流程进行阐述。

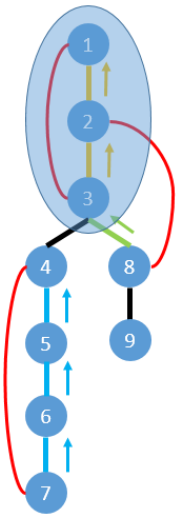


图 24：冗余问题的解决方案

4.1.5 并查集

并查集思想

并查集是一种树形的数据结构，它用于处理一些不交集（集合之间不存在交集）的合并及查询问题，支持以下两种操作：

查找：确定某个元素输入哪个集合

合并：将两个集合合并为一个集合

并查集的重要思想就是，使用集合中的一个元素作为其他所有元素的祖先（包括它自己），然后去代表这个集合。这里使用$fa[i]$表示元素i的祖先，通过一个具体例子来体会并查集两种操作的实际应用。这个例子把集合比喻成帮派，把代表元素比喻成帮主，假设有一开始有5个帮派，每个帮派都只有一个人，那么自己就是帮主，即$fa[i]=i$，如图25所示。

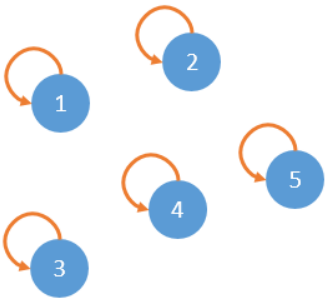


图 25：并查集例子（1）

现在假设有一些帮派想要结盟，比如帮派1和帮派2，然后认帮主1作为新帮主（谁做新帮主并不重要，这里只是想体现它们的集合被合并了），那么修改节点2的祖先$fa[2]=1$（即图中的边）；同理，帮派3和帮派4也想要结盟，把帮主3作为新帮主，修改节点4的祖先$fa[4]=3$，如图26所示。

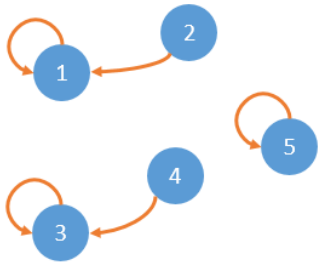


图 26：并查集例子（2）

之后如果节点 1 节点 2 组成的新帮派和节点 3 节点 4 组成的新帮派想要结盟，那么指认的新帮主必须为 1 或 3 之一，这里指定为 1，然后修改 3 的祖先$fa[3]=1$，如图 27 所示。

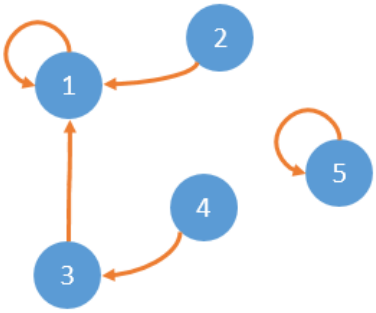


图 27：并查集例子（3）

此时，若帮主 5 询问大侠 4 的帮主，那么大侠 4 需要先询问大侠 3，大侠 3 询问大侠 1，之后就知道大侠 4 的帮主为 1，就是一个不断往上查找的过程，查找过程如图 28 所示。

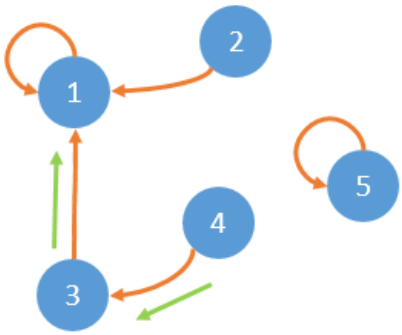


图 28：并查集例子（4）

并查集路径压缩

在上述并查集的例子中，查找节点 4 所在集合的代表元素需要往上查找两步，当集合的大小规模变大时，这种查找方式会使得查找的效率极其低下，因为每一次查找节点 4 所属集合的代表元素都需要往上查找两次。

可以在第一次查找完节点 4 的最终祖先之后，直接把节点 4 的祖先改为其最终祖先，如图 29 所示，在之后查找节点 4 的最终祖先时，就可以直接获得了，大幅度提高了查找效率。

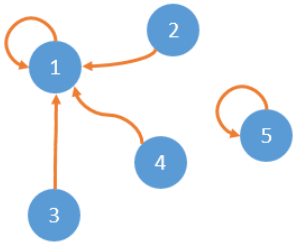


图 29：路径压缩

4.1.6 并查集优化基本思想

回到之前的问题，就是想要把每一个非树边形成的环压缩为一个点来表示，这一步可以使用并查集维护，把节点考虑为集合，每一次考虑非树边就是把该边形成的环中的所有集合合并起来，并且把合并之后的集合的代表元素作为压缩之后的点。

4.1.7 LCA+并查集优化算法流程

以图 30 所示例子叙述算法流程，其中红色边为非树边。

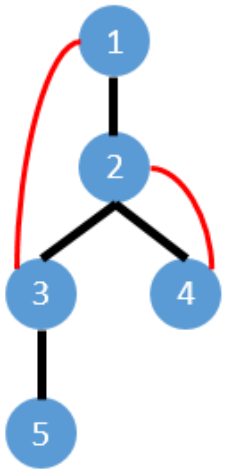
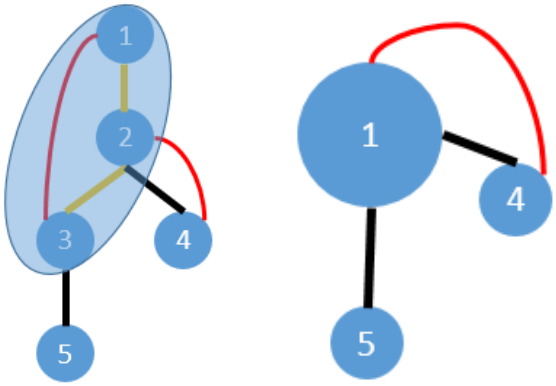


图 30：例子

首先考虑非树边(1,3)，找出该边的两个顶点 1 和 3 所属集合的代表元素 1 和 3（一开始代表元素都为自身），再对两个代表元素寻找 LCA（节点 1），并标记环边，标记环边之后，对该环的所有顶点进行压缩。如图 31(a)所示，将节点 1、2、3 使用并查集合并，并把节点 1 作为合并后集合的代表元素。顶点压缩之后的如图 31(b)所示，注意非树边(2,4)并没有改变，只是因为 2 包括在集合里所以这样连。此时只需要把节点 1、2、3 当做集合看待即可，因为集合中的边已经全部标记过了，对于之后的其他非树边，如果该非树边的一个顶点落在集合中的话，它形成的环会与集合中的边有交集，这些边并不需要再次标记，故只需要考虑集合外的边。

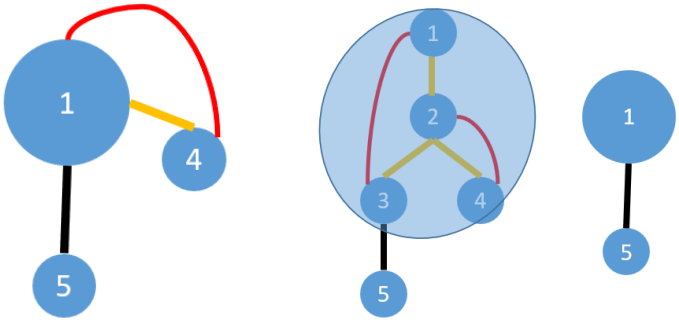


(a) (b)

图 31：算法过程（1）

然后考虑非树边(2,4)，此时只需要标记集合外的树边(2,4)，如图 32(a)所示。之后把节

点 4 与集合 1 合并，并用节点 1 作为合并后的集合的代表元素，如图 32(b)所示，故最终的桥边为(3,5)。



(a) (b)

图 32：算法过程（2）

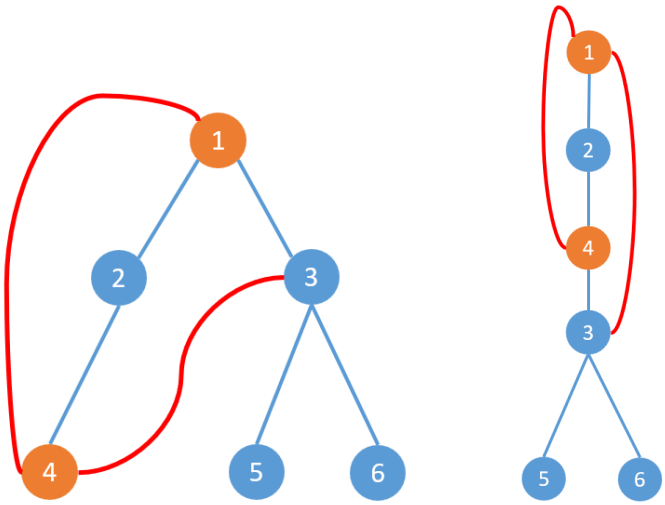
4.1.8 算法复杂度分析

路径压缩之后的并查集合并操作与查询操作都可近似看作$O(1)$。算法运行过程中，需要枚举非树边，之后每一条环边都会在找 LCA 的时候被标记且仅被标记一次，故总共会访问$O(m)$条边与$O(n)$个点。故此算法总时间复杂度为$O(n+m)$。

4.2 差分优化求桥

4.2.1 算法思想

在进行实验的过程中，我发现，图中每一条非生成树边的两个顶点的关系只可能是祖先与孩子的关系。可以使用反证法证明这个结论，如图 33(a)所示，如果存在一条非树边连接了节点 3 和节点 4，那么这个图的生成树就不会这样生成，而是会像图 33(b)一样生成，那么就与原来的树是生成树矛盾了。



(a) (b)

图 33：结论证明

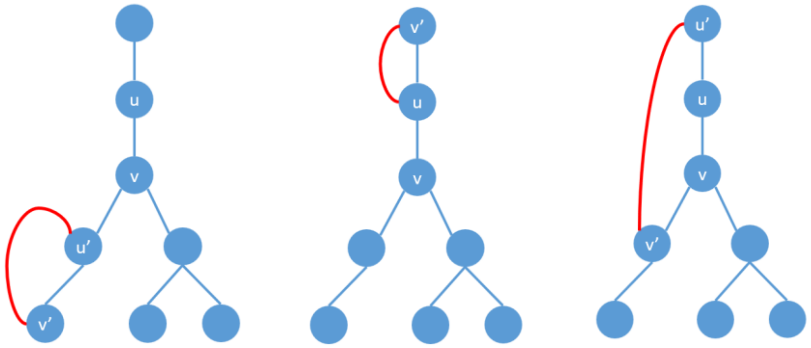
有了这个结论之后，就可以进行以下的考虑了。考虑某一条树边(u,v)（假设u是父亲，

v是孩子），那么以该边作为中心，所有的非树边(u', v')只有以下三种情况：

u', v'都在以v为根的子树中，如图 34(a)所示

u', v'都不在以v为根的子树中，如图 34(b)所示

u', v'有一个在以v为根的子树中，另一个不在以v为根的子树中，如图 34(c)所示



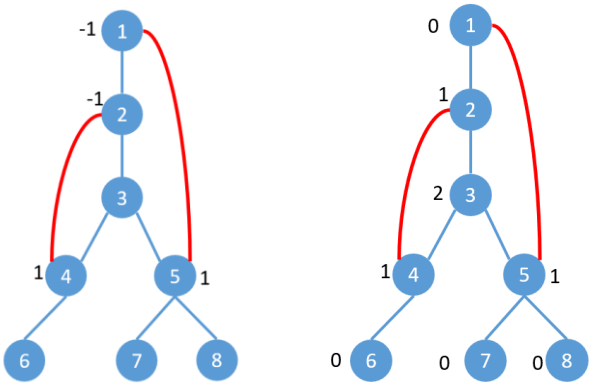
(a) (b) (c)

图 34：非树边的三种情况

可以看到，前两种情况的非树边不会对判断树边(u, v)是否为桥产生影响，因为它们无法与树边(u, v)产生环。故判断树边(u, v)是否为桥，只需要判断是否存在第三种情况的非树边，接下来对如何判断此类非树边提出解决方案。

4.2.2 差分解决方案

可以利用一种差分的技术，将每一条非树边深度较大的点加上-1 的标记，深度较小的点加上+1 的标记，最后对生成树中所有子树（以某个顶点为根）中的标记求和，以v为根的子树中标记的和就表示了相对于树边$(fa[v], v)$的第三类树边数量（$fa[v]$为v在生成树中的父亲）。那么也就是说，只有求和之后为 0 的节点v，树边$(fa[v], v)$才为桥边。如图 35 所示例子，对非所有非树边打上标记之后如图 35(a)所示，对每个节点的子树标记求和之后如图 35(b)所示，故桥边只有(4,6)(5,7)(5,8)。



(a) (b)

图 35：对非树边打上差分标记

4.2.3 差分算法流程

算法流程比较简单，首先生成原图的生成树（生成森林），然后枚举所有的非树边，把两端的节点按照深度大小打上对应的标记。之后对打完标记的生成树进行子树标记求和，这

个操作只需要 dfs 整颗生成树，在回溯的时候更新某个节点的标记之和即可。

4.2.4 算法复杂度分析

首先需要枚举每一条非树边并打上标记，需要$O(m)$的时间复杂度。之后需要遍历一遍生成树求出标记总和，需要$O(n+n-1)=O(n)$的复杂度，故总时间复杂度为$O(n+m)$。

5 相关实验测试算法效率

5.1 高效算法的正确性测试

对于两个高效算法，生成了若干组点数与边数均为 1000 的数据进行算法正确性测试。比较基准法与高效算法的运行结果，全部运行结果均一致。

5.2 各种算法在给定数据集上的效率测试

实现了上述所有算法并在给定的两个数据集上进行实验测试运行效率，测试结果如表 1 所示。由第 2 节两种基准法的分析得，基准法 2 在理论上常数较基准法 1 小，实际实验下来确实证明了这一点。生成树优化之后结果也符合理论分析的结果，在生成树优化之后，$O(m^2+nm)$的复杂度变为$O(n^2+nm)$，MediumG 的边数比点数大，故运行时间会有明显的下降。可以看到，两种高效算法在 MediumG 图上的运行效率都高于之前的方法。

在 LargeG 图（1000000 个点,7800000 条边）上，除了两个高效算法，其他的算法都无法在有限时间内得出结果，可以看到两个高效算法的运行时间都比较可观。

表 1：不同算法在给定数据集上的实际运行时间(ms)比较

算法	MediumG	LargeG
基准法 1	6.464	>100000
基准法 2	5.618	>100000
基准法 1+生成树优化	2.678	>100000
基准法 2+生成树优化	2.703	>100000
LCA+并查集优化	0.341	3870.184
差分优化	0.039	2806.167

5.3 随机生成数据对基准法 1+生成树优化算法进行测试

我固定边数为 50000，改变点数（从 1000~5000），测试结果如表 2 所示，理论时间以点数为 1000 时的实际运行时间为基准，由于此时算法复杂度为$O(nm+n^2)$，且此时边数远多于点数，故n^2项影响极小，且此时边数不变，故计算理论值时使用n作为变量计算。从图 36 中可以看到理论值与实际值拟合较好，且符合与点数的线性关系。

表 2：基准法 1+生成树优化算法在不同点数规模下的运行效率测试

数据规模（点数）	1000	2000	3000	4000	5000
理论运行时间(s)	2.712	5.425	8.137	10.850	13.562
实际运行时间(s)	2.712	5.502	8.358	11.321	14.411
误差	0.00%	1.42%	2.71%	4.34%	6.26%

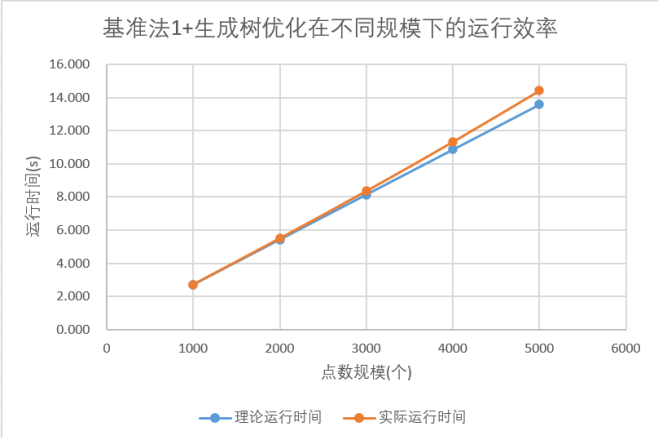


图 36：基准法 1+生成树在不同点数规模下的运行效率

然后，又让边数与点数相等，从 2000 个点到 10000 个点，测试其效率随规模的变换，测试结果如表 3 所示，可以看到，实际运行时间与理论运行时间的误差较大。此时由于边数与点数相同，故算法的复杂度为$O(nm + n^2) = O(n^2)$，但因为常数较大故误差较大。运行效率变化图如图 37 所示，曲线基本满足二次函数的曲线增长趋势。

表 3：基准法 1+生成树优化算法在不同规模下的运行效率测试

数据规模（点数与边数）	2000	4000	6000	8000	10000
理论运行时间(s)	0.160	0.639	1.439	2.558	3.997
实际运行时间(s)	0.160	0.575	1.297	2.891	5.042
误差	0.00%	-10.09%	-9.87%	13.01%	26.16%

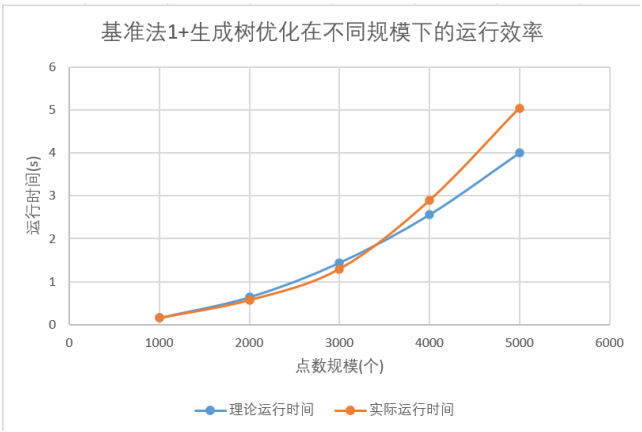


图 37：基准法 1+生成树在不同规模下的运行效率

5.4 随机生成数据对 LCA+并查集优化的高效算法进行效率测试

首先固定点数为 5000，改变边数从 100000~500000，测试算法运行效率随边数规模的变化，测试结果如表 4 所示。可以看到，理论运行时间与实际运行时间拟合较好。其运行效率随不同规模数据的变化图如图 38 所示，可以看到，基本上拟合了以边数为基准的线性增长

趋势。

表 4：LCA+并查集优化在不同数据规模下的运行效率

数据规模（边数）	100000	200000	300000	400000	500000
理论运行时间(ms)	23.746	47.492	71.238	94.984	118.730
实际运行时间(ms)	23.746	46.014	70.257	93.935	119.260
误差	0.00%	-3.11%	-1.38%	-1.10%	0.45%

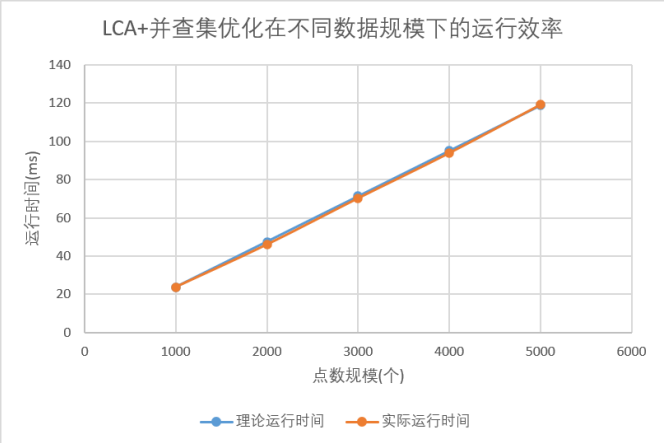


图 38：LCA+并查集优化在不同规模下的运行效率

5.5 随机生成数据对差分优化的高效算法进行效率测试

仍然使用 5.4 节的数据对差分优化的高效算法进行效率测试，测试结果如表 5 所示。可以看到，理论运行时间与实际运行时间拟合较好。其效率变化曲线如图 39 所示，可以看到，基本上拟合了以边数为基准的线性增长趋势。

表 4：LCA+并查集优化在不同数据规模下的运行效率

数据规模（边数）	100000	200000	300000	400000	500000
理论运行时间(ms)	16.663	33.326	49.989	66.652	83.315
实际运行时间(ms)	16.663	33.725	49.484	67.222	83.575
误差	0.00%	1.20%	-1.01%	0.86%	1.51%

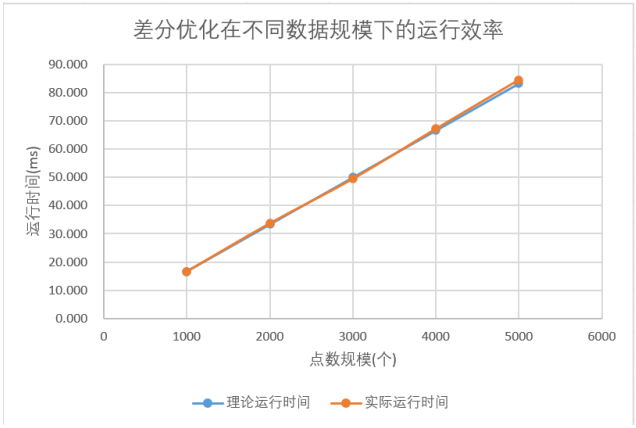


图 39：差分优化在不同规模下的运行效率

四、实验心得

详细分析了问题，并提出多种解决方案，对基准算法进行详细阐述并提出优化方案。使用并查集设计了一个高效算法，得出了可观的实验结果，同时使用差分技术设计了一个比并查集更高效的算法。进行了较多实验对时间复杂度进行验证。

掌握了桥的求解以及并查集的应用。

一开始在跑给定的 largeG 数据时，程序一直会崩溃，在查阅相关资料之后，发现这是因为运行时递归层数较多，故运行时栈会爆满。故需要手动使用命令行参数把运行时的栈空间变大。

五、附件说明

- **code（实验所编写代码）**
  - **basic.cpp（两种基准法）**
  - **basic+SpanningTree.cpp（带生成树优化的基准法）**
  - **Different.cpp（差分优化）**
  - **generate_data.cpp（随机数据生成器）**
  - **LCA.cpp（LCA+并查集优化）**
- **result（实验结果数据表）**
- **PPT（演讲 PPT）**
- **PDF（实验报告 PDF 版本，更好的阅读体验）**

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。