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：图论—求桥

学院：计算机与软件学院

专业：计算机科学与技术

指导教师：杜智华

报告人：欧阳宇杰 学号：2021150143 班级：计科 2 班

实验时间：2023 年 5 月 16 日~5 月 22 日

实验报告提交时间：2023 年 5 月 23 日

教务部制

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

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。

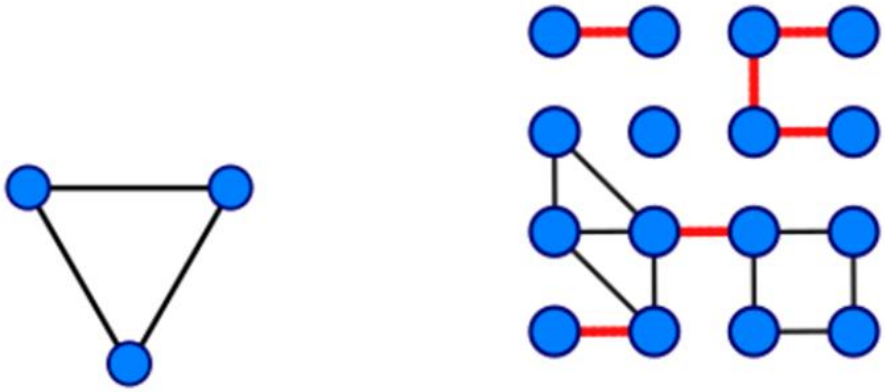


图 1 实验例图

求解问题

找出一个无向图中所有的桥。

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、实验要求

1、实现上述基准算法。

2、设计的高效算法中必须使用并查集，如有需要，可以配合使用其他任何数据结构。

3、用图 2 的例子验证算法正确性。

4、使用文件 mediumG.txt 和 largeG.txt 中的无向图测试基准算法和高效算法的性能，记录两个算法的运行时间。

5、设计的高效算法的运行时间作为评分标准之一。

6、提交程序源代码。

7、实验报告中要详细描述算法设计的思想，核心步骤，使用的数据结构。

四、实验过程与步骤

（一）基准算法

❖ 思想：

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。因此，我们可以比较直接地想到这样的一种算法：

1. 先用 DFS 统计图的连通分量个数 num1。
2. 接着遍历边集中的每条边 e，在图中删除该被遍历的边 e。
3. 然后再次用 DFS 获取整张图的连通分量个数 num2。
4. 如果 num2>num1，说明边 e 为桥。如果 num1>num2，则说明边 e 不是桥。
5. 将被删除的边 e 重新加入图中，然后从第二个步骤开始重复，直到边集中所有的边都已经被遍历。

❖ 伪代码描述如下：

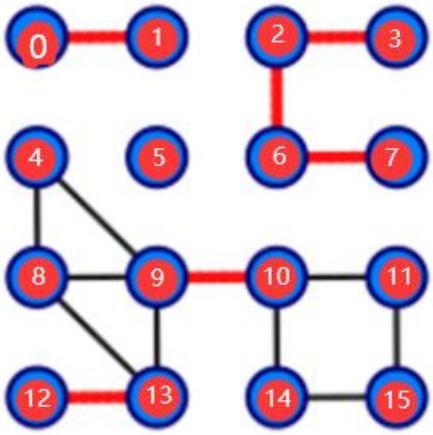
```
Basic_DFS(graph)
    num1 = connection_count(graph)
    对于每一条边(u, v):
        将 (u, v) 从图中移走
        num2 = connection_count(graph)
        if num2 > num1
            则(u,v) 是桥
        将边 (u, v) 重新加进图中
```

```
connection_count(graph):
    Count = 0
    For every vertex (v):
        If visited[v] = false:
            DFS (v)
            Count++
    return Count
```

```
DFS(v):
    visited[v] = true
    for 每一个相邻点adj of v:
        if visited[adj] = false:
            DFS (adj)
```

❖ 验证算法正确性

题目要求用第二个实验例图验证算法正确性，故我对第二个实验例图各点编号如左下图，得到结果如右下图所示，可以看到得到了正确的桥和桥的数量。



❖ 时间复杂度分析：

设图中顶点个数为 n，边数为 e。在使用邻接表的情况下，一次 DFS 遍历的时间复杂度为 $O(n+e)$。又由上面分析得对于每一条边，算法都需要用一次 DFS 来计算新的连通分量。所以一共需要进行 e 次 DFS 操作，因此算法的总时间复杂度为: $O(e^2+ne)$。

假设对于稀疏图，边数 e 的个数∝顶点个数 n。故算法时间复杂度可以表示为 $O(n^2)$。

假设对于稠密图，边数 e 的个数∝顶点个数的平方n^2。故算法时间复杂度可以表示为 $O(n^4)$。

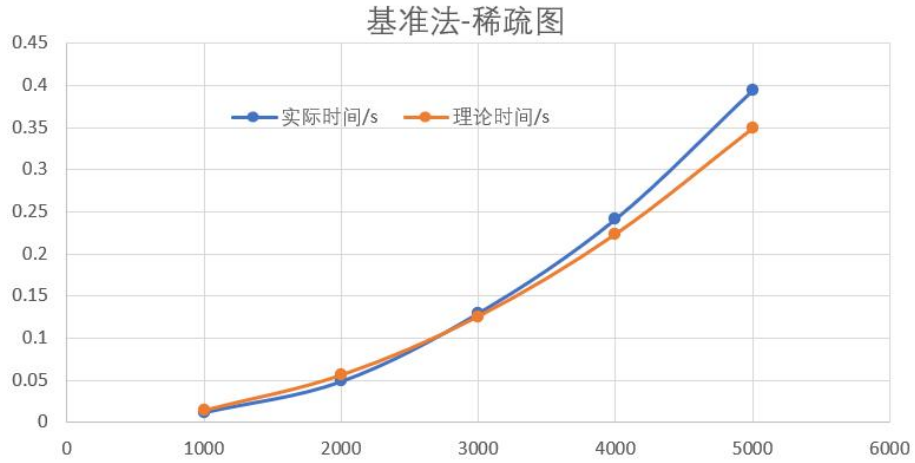
❖ 数据分析：

➢ 对于稀疏图 $e \propto n$：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：

其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	1000	2000	3000	4000	5000
实际时间/s	0.012	0.049	0.13	0.242	0.395
理论时间/s	0.014	0.056	0.126	0.224	0.35
相差比例	-0.166667	-0.1429	0.03077	0.07438	0.11392

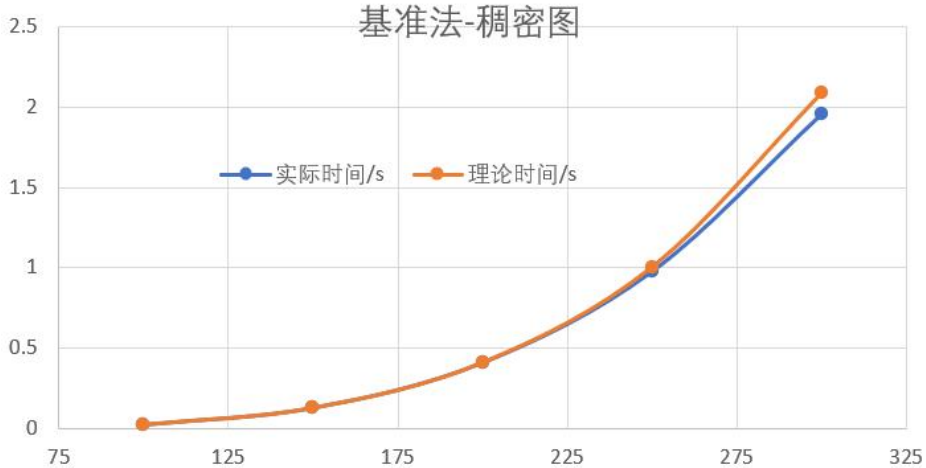


可以看到基本拟合，说明在稀疏图中基准法符合 O(n^2)的时间复杂度。

➢ 对于稠密图 e $\propto n^2$：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：
其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	100	150	200	250	300
实际时间/s	0.027	0.131	0.412	0.979	1.957
理论时间/s	0.0258	0.13035	0.412	1.0058	2.08575
相差比例	0.04444444	0.00496	0	-0.0274	-0.0658



可以看到很拟合，说明在稠密图中基准法符合 O(n^4)的时间复杂度。

（二）算法 2—基准法+并查集求连通分量+生成树优化

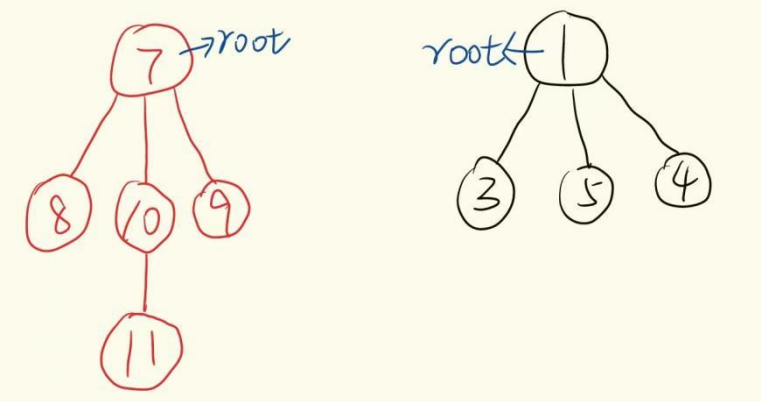
在基准法的基础上运用并查集来计算连通分量,并采用生成树来进行优化。总的来说，和基准法思路相同，通过删除边并计算连通分支数目来查找桥。只在下面两方面进行了修改：

1. 运用并查集来计算连通分量
2. 只尝试删除生成树上的边，因为只有生成树上的边才有可能是桥

✧ 并查集

并查集是一种树型的数据结构，用于处理一些不相交集合的合并及查询问题。

并查集的重要思想在于，用集合中的一个元素代表集合。将这个集合画成一棵树则代表集合的元素就是树的根节点。如下图中红色节点和黑色节点就是两个不同集合。



并查集有查询和合并两个常见操作：

❖ 查询

查询即查询当前两元素是否在同一集合中。通过递归层层向上访问，直至访问到根节点，若两元素根节点相同，则两元素在同一集合中。否则不在同一集合中。

以下是查找一个节点的根节点的伪代码：

```
Get_Parent(x)
    if(parent[x] == x)
        return x;
    else
        return Get_Parent(parent[x]);
```

❖ 合并

合并即将两个集合合并为一个集合。只需先通过查询，查询出一个集合的根节点，并修改根节点指向另一集合即可。

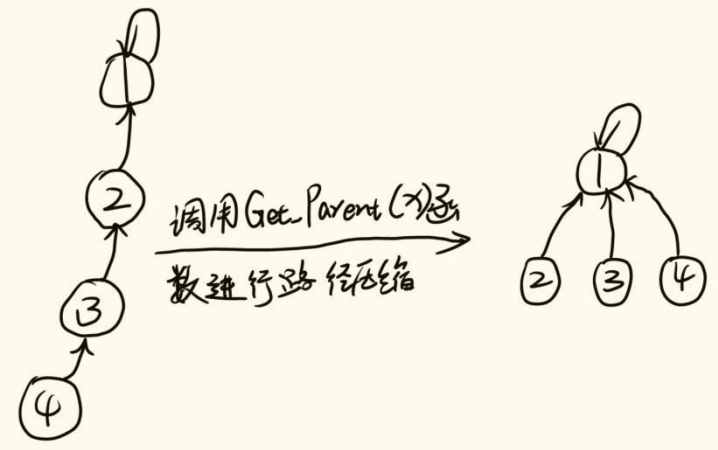
```
Union(i, j)
    parent[Get_Parent(i)] = Get_Parent(j);
```

➢ **路径压缩**

路径压缩是并查集常用的优化。随着链越来越长，我们想要从底部找到根节点就要经过越来越多的节点，这会导致查询变慢。可以采用路径压缩来优化。既然只关心元素对应的根节点，那就将每个元素都直接指向根节点就好了。实现的方式是在查询的过程中，把沿途的每个节点的父节点都设为根节点。这样在下一次再查询时，可以直接到达根节点，大大减少了查询的时间复杂度。

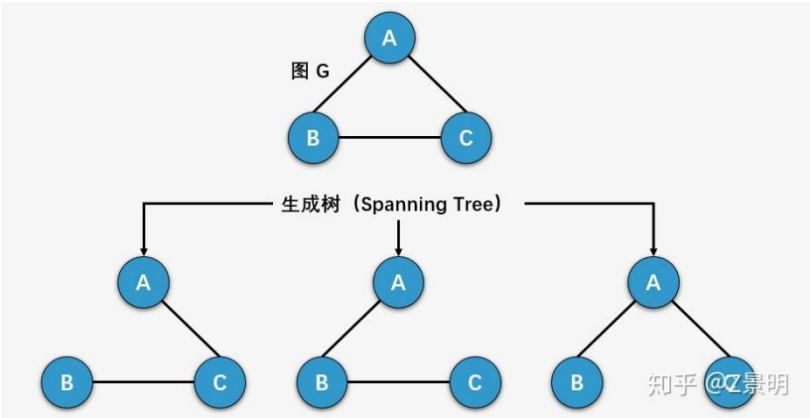
```
Get_Parent(x)
    if (x == parent[x])
        return x;
    else
        parent[x] = Get_Parent(parent[x]);//父节点设为根节点
    return parent[x]; //返回父节点
```

下面是一个路径压缩的例子：



✧ 为什么桥一定在生成树上？

一个连通图的生成树是一个极小的连通子图，它包含图中全部的 n 个顶点，但只有构成一棵树的 n-1 条边。



对于图的每一个连通分量而言，其对应的生成树有这个性质：位于生成树的边有可能是桥，但不在生成树上的边一定不是桥。因为根据树的性质：树是边数最小的无环图。所以向生成树上添加任意不在生成树上的边，必定会形成环。也

就是说不在生成树上的边一定是环边。那么又根据桥的定义环边一定不是桥。因此只需要判断生成树上的边是否为桥便可以了。

则根据思路可写出如下的算法伪代码：

```
algorithm2()
    before = connection_count();    // 记录初始图连通分支数目
    visit_init();  //初始化vis数组
    tree_edges;  //生成树边集
    for (i = 0; i < n; i++) // 通过dfs获得生成树边集
        if (vis[i] == 0)
            dfs_getTree(i, -1, tree_edges);
    for (t = 0; t < tree_edges.size(); t++) //遍历生成树边集
        disjointSets_init();    // 并查集初始化
        cnt = n;  //连通分量初始化为节点个数
        for (i = 0; i < edges.size(); i++) //遍历边集来合并集合
            if (edge_equal(tree_edges[t], edges[i])) // 删除第t条边
                continue;//用continue来不让其用于合并集合,从而"删除"
            f1 = query_compression(edges[i][0]);
            f2 = query_compression(edges[i][1]);
            if (f1 != f2) father[f1] = f2, cnt--;   // 合并两个集合
        if (cnt > before)
            brinum++;
```

```
void dfs_getTree(cur, pre, vector<vector<int>>& t)
{
    //如果cur不是根节点,就将<pre, cur>加进生成树边集
    if (pre != -1) t.push_back(vector<int>{pre, cur});
    vis[cur] = 1;
    for (auto cur_adj:adj[cur])
        if (vis[cur_adj] == 0)
            dfs_getTree(cur_adj, cur, t);
    //又从cur的邻边开始dfs
}
```

```
int query_compression(int x)
{
    if(father[x]==x) return x;
    int res = query_compression(father[x]);
    father[x] = res; //就是查询并且路径压缩
    return res;
}
```

❖ 验证算法正确性

题目要求用第二个实验例图验证算法正确性，得到结果如下图所示，可以看到得到了正确的桥和桥的数量。

❖ 时间复杂度分析：

外层循环为生成树边数，生成树边数接近于顶点个数，故外层循环时间复杂度为 O(n)。内存循环次数为边集个数 e

假设对于稀疏图，边数 e 的个数∝顶点个数n。故算法时间复杂度可以表示为 $O(n^2)$。

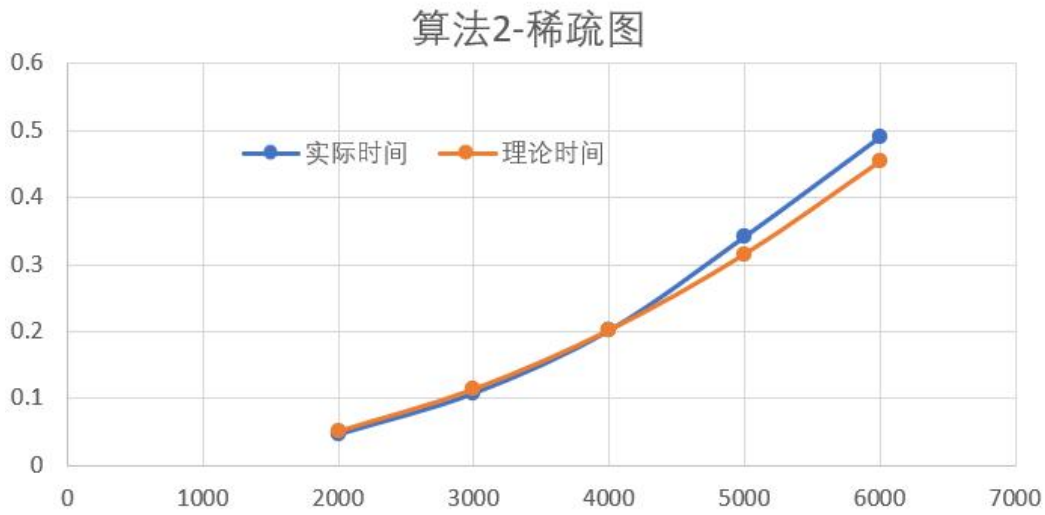
假设对于稠密图，边数 e 的个数∝顶点个数的平方n^2。故算法时间复杂度可以表示为 $O(n^3)$。

❖ 数据分析：

➢ 对于稀疏图 e ∝ n：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：
其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	2000	3000	4000	5000	6000
实际时间/s	0.0463	0.1078	0.202	0.342	0.491
理论时间/s	0.0505	0.11363	0.202	0.315625	0.4545
相差比例	-0.0907	-0.054	0	0.07711988	0.07434



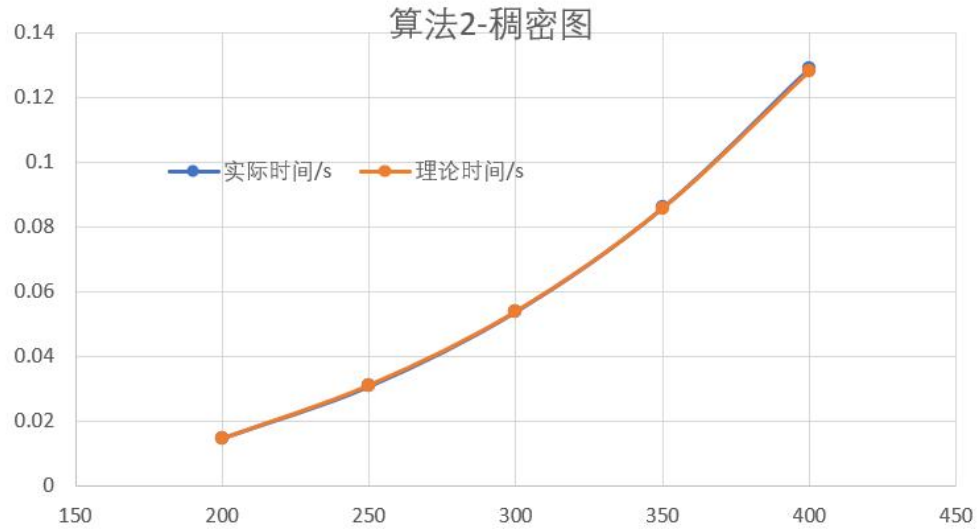
可以看到基本拟合，说明在稀疏图中算法 2 符合 $O(n^2)$的时间复杂度。

➢ 对于稠密图 e ∝ n^2：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：

其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	200	250	300	350	400
实际时间/s	0.015	0.031	0.054	0.086	0.129
理论时间/s	0.014815	0.03125	0.054	0.08575	0.128
相差比例	0.012347	-0.0081	0	0.00290698	0.00775



可以看到严格拟合，说明在稠密图中算法 2 符合 O(n^3)的时间复杂度。

（三）算法 3—最终高效算法

根据桥的定义，“一条边是一座桥当且仅当这条边不在任何环上”。故桥边的计算还可以通过排除法得到，即总边数 - 环边数。则问题就转化为了求图中的环边。而且首先非生成树边一定是环边（在前面已经说了）。

算法思路：首先通过 dfs 获得生成森林，然后依次将非生成树边加入生成树之中，如果加入非生成树边之后形成了环，则将未标记的环边进行标记。最后得到环边总数。桥数 = 总边数 - 环边数。

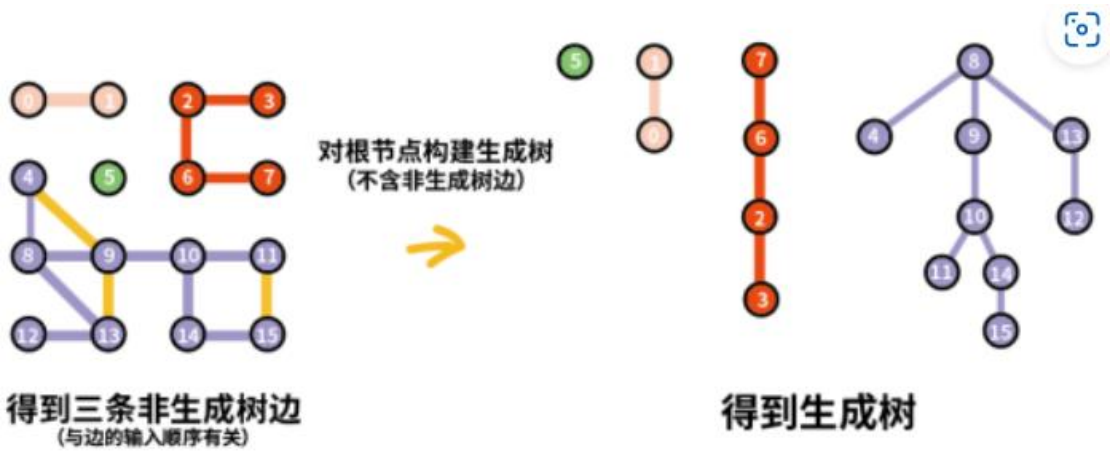
此处通过求 LCA 算法来求图中的环边。

❖ 求 LCA 算法

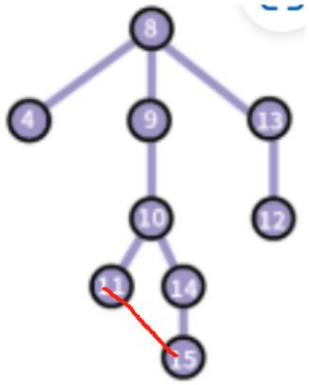
求 LCA 算法用于求两节点的共同祖先，如果两个结点拥有相同的祖先，则必能形成闭环（我们在下面的算法思路讲解中给出求 LCA 具体步骤）。

➢ 算法思路讲解

■ 我们先通过 dfs 获得生成森林，如下图所示：



- 得到生成树之后，我们需要将每条非生成树边依次加入生成树。对于这条边的两个节点，寻找它们的最小公共祖先节点，并且将寻找过程中遇到的边标记为环边。下面我们拿（11，15）这条边进行解释。



显然 11-10-14-15 可以形成闭环，在这颗生成树中 11、15 的公共祖先为 10，向上遍历的过程中 10-11，14-15，10-14 均被标记为环边。对于其他非生成树边同理。

求 LCA 具体步骤：（以边 11-15 为例）

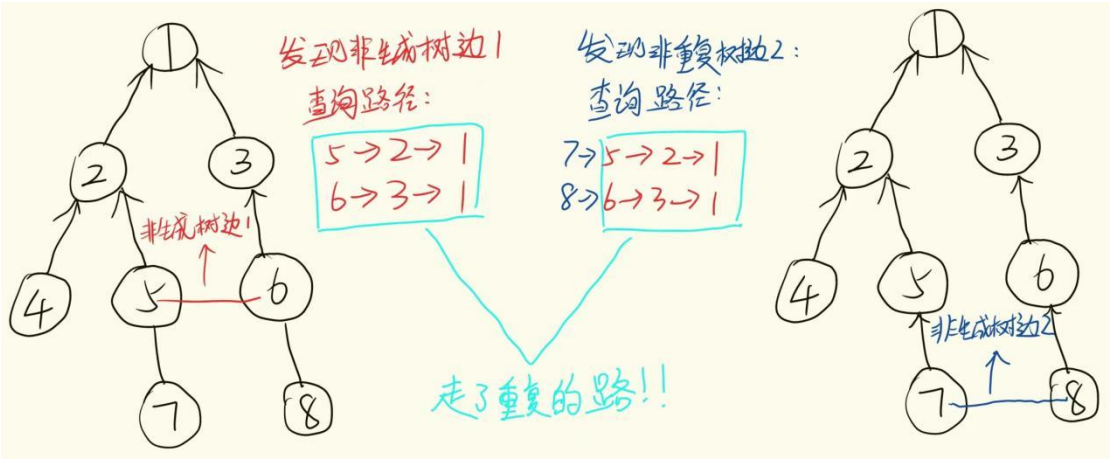
(1) 先同步高度。11 高度：4，15 高度：5。首先需要对节点 15 沿着父亲节点向上找，直到其高度为 4（与 11 的高度一样）。于是我们首先找到 14-15 边。
(2) 此时节点 11 和节点 14 的高度都为 4。它们一起寻找它们的父亲节点 10，记录下来 10-11 边和 10-14 边。
(3) 若找到共同父亲节点后结束搜索。上图例子中找到节点 10 就结束了搜索。但在别的情况下若没有找到共同父亲节点则两节点继续寻找它们的父亲节点直到找到共同父亲节点。

上图例子最终找到四条环边。

❖ 还可以用路径压缩优化算法

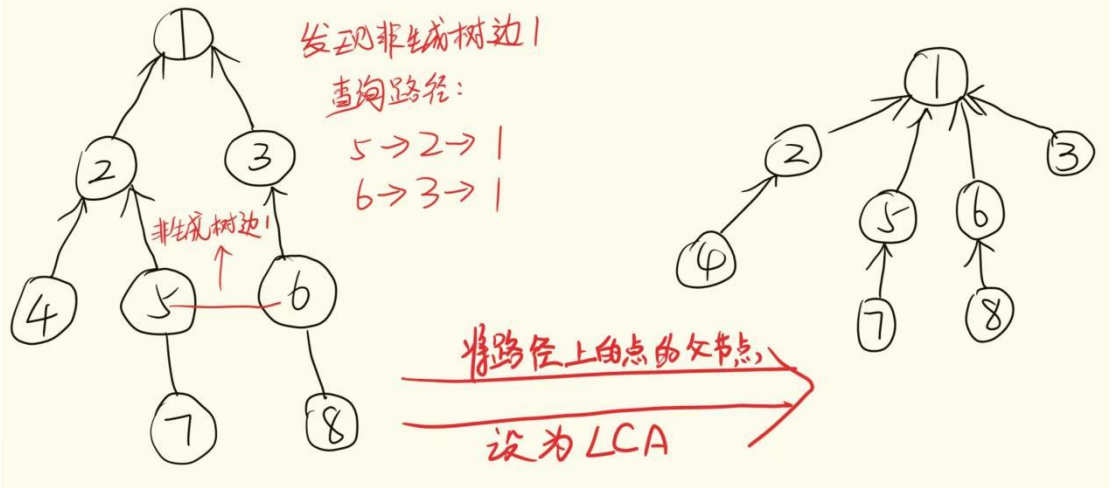
有时因为边和非成树边非常多，我们可能会走很多重复的路。

如下就是一个简单的例子：



我们在查找非生成树边 1 的两点时，走过了 5->2->1 路径和 6->3->1 路径，并且也将（2，5）、（1，2）、（3，6）、（1，3）边加入了环边集即标记为了环边。可我们在在查找非生成树边 2 的两点时，又要走 7->5->2->1 路径和 8->6->3->1 路径，可 5->2->1 路径和 6->3->1 路径我们之前已经走过了，相应路径上的边也已经标记过了，我们走了重复路！！！故我们可以通过使其不走重复路来改进算法。

我们可以在沿着生成树向上找 LCA 的时候，找到 LCA 并记录它，然后将路径上的所有节点的父节点都指定为 LCA（即查找的终点），这也表示节点和 LCA 之间的路径被标记过了，在下次向上查找的时候，我们会直接跳到 LCA 上，就不用走重复路了。



根据上面阐述的思路可以写出如下的算法伪代码：

```
void efficient()
    for (i = 0; i < n; i++) // 生成生成树
        if (!vis[i])
            dfs_gettree(i, i, 0);
    visit_init();
    for (i = 0; i < n; i++)
        tree_edges.insert(vector<int>{tree[i], i});
    for (i = 0; i < edges.size(); i++)  //遍历边集
        if (tree_edges.find(edges[i]) == tree_edges.end())  //如果被遍历到的边不在生成树上
            p1 = edges[i][0], p2 = edges[i][1];
            cirEdges.insert(vector<int> {p1, p2});  // 标记该边为环边
            if (depth[p1] < depth[p2])
                swap(p1, p2);   // 默认p1为深的点
            while (depth[p1] > depth[p2]) //同步高度并加边进入环边集
                pre1 = p1, p1 = tree[p1];
                cirEdges.insert(vector<int> {p1, pre1});
            while (tree[p1] != tree[p2]) // 找LCA并加边进入环边集
                pre1 = p1, pre2 = p2;
                p1 = tree[p1],p2 = tree[p2];
                cirEdges.insert(vector<int> {p2, pre2});
                cirEdges.insert(vector<int> {p1, pre1});
            pre1 = p1, pre2 = p2;
            p1 = tree[p1],p2 = tree[p2];
            cirEdges.insert(vector<int> {p2, pre2});
            cirEdges.insert(vector<int> {p1, pre1});
            //压缩策略,将路径上所有点父节点设为LCA,并设置好深度
            path_compression(tree[p1],edges[i][0]);
            path_compression(tree[p2],edges[i][1]);
    最后，遍历边集,如果边不是环边就说明是桥,输出或桥数量+1即可
```

```
path_compression(LCA,p)
    while(tree[p]!=LCA)
        ptemp=tree[p]
        tree[p]=LCA;
        depth[p]=depth[LCA]+1;
        p=ptemp;
```

❖ 验证算法正确性

题目要求用第二个实验例图验证算法正确性，得到结果如下图所示，可以看到得到了正确的桥和桥的数量。

❖ 时间复杂度分析

循环次数为边个数 e。遇到非生成树边就要查找，从非生成树边的两点 p1 和 p2 找 LCA，查找的最差时间复杂度为 O(n)，然后从 p1 和 p2 开始路径压缩，路径压缩的最差时间复杂度也为 O(n)。所以最差时间复杂度是

$$T = O(e) * (O(n) + 2 * O(n)) = O(en)$$

然而，对于大数据量级下的查找操作，经过了并查集的路径压缩，实际上很快大部分要查找的节点其父节点就已经被设置为了 LCA。这样查找操作的时间复杂度会接近 O(1)，且这样路径压缩就无需进行。则此时的时间复杂度为

$$T = O(e) * O(1) = O(e) \qquad n\ and\ e\ are\ large$$

假设对于稀疏图，边数 e 的个数∝顶点个数n。则算法时间复杂度可以表示为 $O(n)$。

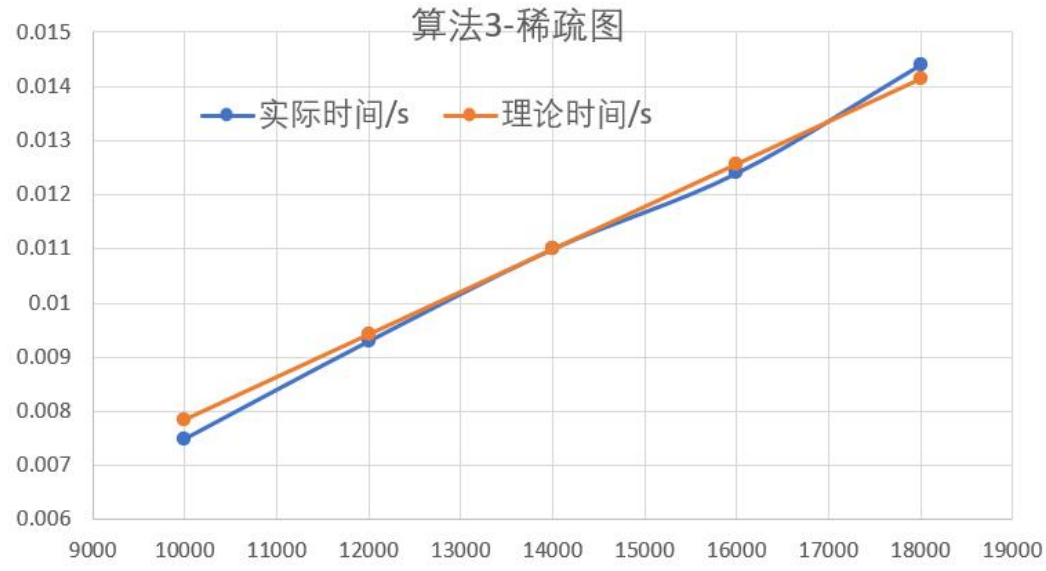
假设对于稠密图，边数 e 的个数∝顶点个数的平方n^2。则算法时间复杂度可以表示为 $O(n^2)$。

❖ 数据分析：

➢ 对于稀疏图 e ∝ n：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：
其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	10000	12000	14000	16000	18000
实际时间/s	0.0075	0.0093	0.011	0.0124	0.0144
理论时间/s	0.00786	0.00943	0.011	0.01257	0.01414
相差比例	-0.0476	-0.0138	0	-0.0137097	0.01785

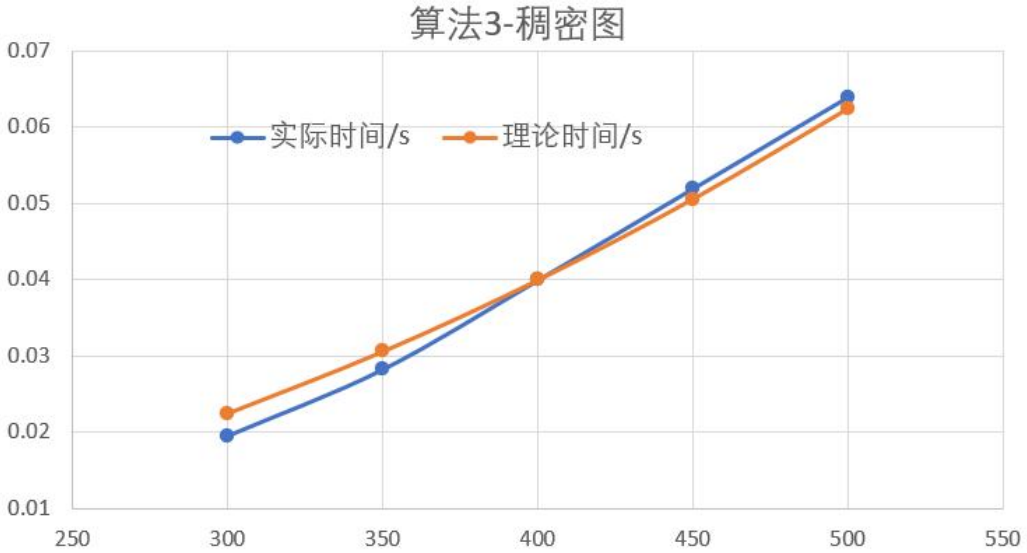


可以看到基本拟合，说明在稀疏图中算法 3 符合 O(n)的时间复杂度。

➢ 对于稠密图 e ∝ n^2：

生成多组随机数据，每组数据运行 10 次并取平均值后做表作图如下：
其中相差比例为（实际时间-理论时间）/实际时间。

节点个数	300	350	400	450	500
实际时间/s	0.0196	0.0283	0.04	0.052	0.064
理论时间/s	0.0225	0.030625	0.04	0.05063	0.0625
相差比例	-0.148	-0.0821555	0	0.02644	0.02344



可以看到基本拟合，说明在稠密图中算法 3 符合 O(n^2)的时间复杂度。

✧ 最后根据实验要求运行 mediumDG.txt 和 large.Gtxt 数据集

mediumDG.txt：

运用算法种类	基准法	算法 2	算法 3
运行时间	0s	0s	0s

largeG.txt: 仅有算法 3 能够得到正确结果如下图：

```
读取图数据完毕
顶点数量: 1000000  边数量: 7586063
5.307s
658123 -- 724640 是桥
639238 -- 969090 是桥
467595 -- 630627 是桥
467595 -- 907820 是桥
461822 -- 548437 是桥
317390 -- 589095 是桥
148837 -- 372243 是桥
95760 -- 773903 是桥
桥个数: 8
```

则有右表：

数据集：largeG.txt	运用算法 3
运行时间	5.307s

五、实验总结与体会

1. 求解问题时可以考虑用排除法来优化算法，从直接探究问题的所求转变为探求问题所求的反面。如在此次实验中我通过探求不是桥的边（即在环上的边）间接求解了问题所要求的桥边。

2. 在对于图的算法中，要同时考虑到稀疏图和稠密图，算法在稀疏图和稠密图上的时间复杂度通常是不一样的。

3. 对于极大的数据集，IDE 默认给的栈空间是不够的，我们有必要扩大 IDE 的栈空间。如在此次实验中我就将 VS 的栈空间修改为了 50MB 才能成功运行完毕 largeG.txt 数据集并得到正确的结果。

4. 在有关图的问题中可以考虑用生成树/生成森林来优化算法。如在此次实验中我后面实现的两个算法都使用了生成树，这很大的优化了算法的时间复杂度。

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。