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——桥问题

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 5 月 26 日—2024 年 6 月 1 日

实验报告提交时间：2024 年 6 月 1 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：

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。

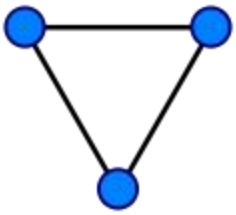

图 1 没有桥的无向连通图

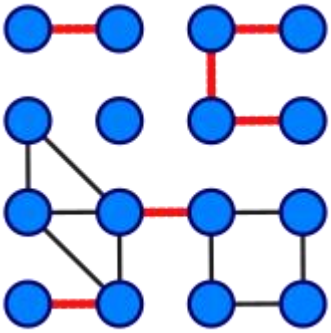

图 2 这是有 16 个顶点和 6 个桥的图（桥以红色线段标示）

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。

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、实验内容及过程：

1. 基准算法

思路：

如果一条边是桥，那么当我们删除这条边后，图的连通块数量会增加，我们可以通过这个性质解题。

基本步骤：

（1）计算原图的连通快的数量。

（2）遍历图的每一条边，一条一条边删除。

（3）删除一条边后，重新计算连通块的数量。

（4）如果连通块的数量增加了就说明这条边是一座桥。

（5）把删除的边复原然后继续遍历。

存储结构：

使用邻接表而不是邻接矩阵，因为实验数据多为稀疏数据，若使用邻接矩阵就会得到系数矩阵，在大多数情况下十分消耗空间，所以我选择了邻接表。时间复杂度上，如果图的顶点数为N，边数为E，那么使用邻接矩阵的时间复制度为O（N^2），使用邻接表的时间复杂度为O（N＋E），在稀疏图的情况下，E<N^2，所以使用邻接表更加高效。

实验给的样例中，也时系数图的结构，所以使用邻接表确实是更加高效。

伪代码：

Algorithm 1：Base

```
Input：None
Output: num_of_bridges
Set(visit)=0
Bridges_old=0
For i in range(n): //计算原本的连通分量数
    If(!visit[i])
        Bridges_old++
        DFS(i) //使用深度优先搜索
For i in range(EDGE)：//遍历每一条边
    Set（visit）=0
    Remove edge[i] //删除边
    Bridges_new=0
      For j in range(n)://计算新的连通分量数
          If(!visit[j])
              Bridges_new++
              DFS(j)
     If(bridges_new>bridges_old)
          Num_of_bridges++
     Add edge[i] //把删除的边加回去
```

说明：先将 visit 数组清零，然后计算原本的连通块数量，之后遍历每一条边，删除遍历到的边后重新计算连通块数，如果块数发生改变，则桥数量加一，最后恢复删除的边。

时间复杂度分析：

顶点数为 n，边数为 e，那么单次深度遍历的时间复杂度为 0（n＋e），由于有 e 条边，那么需要遍历 e 次，所以时间复杂度为 0（en＋e^2），对于稀疏图，e 可以看为 n，所以时间复杂度为 0（n^2），对于稠密图，e 可以看作 n^2，所以时间复杂度为 0（n^4）。

2． 基准算法改进

思路：

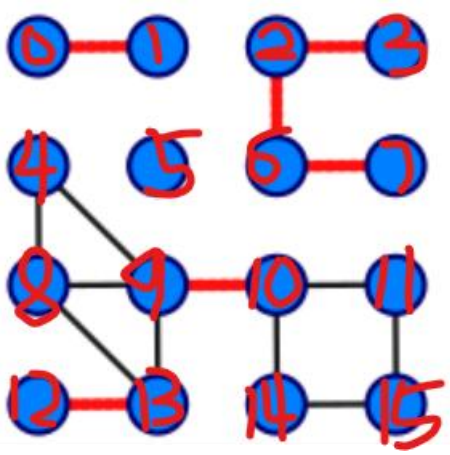

图 3 六桥图

利用原本的基准算法，如果我们删除了（0，1）这条边，那么我们需要遍历整张图求得连通块的数量，这样是很低效的，也是不必要的。我们只需要遍历 0，1 所在的连通子图即可，如果删除了（0，1）这条边后，这个连通子图依旧连通，那么条边就不是桥，反之，这条边就是桥。那么如何判断这个子图是否连通呢，只需要从 0 开始遍历，遍历完成后如果 1 被遍历到了，那么这个子图就依旧是连通子图，反之这个子图就不是连通子图，那么这条边就是桥。

伪代码：

Algorithm 2：Base_improved

Input：None

Output: num_of_bridges

```
1. Set(visit)=0
2. Bridges_old=0
3. For i in range(n): //计算原本的连通分量数
4.     If(!visit[i])
5.         Bridges_old++
6.         DFS(i) //使用深度优先搜索
7. For i in range(EDGE)：//遍历每一条边
8.     Set（visit）=0
9.     Remove edge[i] //删除边
10.    DFS(edge[i].x1）
11.    If(visit[edge[i].x2]==0)//如果不连通，则是桥
12.        Num_of_bridges++
13.    Add edge[i] //把删除的边加回去
```

说明：
主要修改的部分为蓝色字体部分，我们不再需要遍历整张图求解连通块的数量，只需要从被删除边的一个端点开始遍历，然后观察另一个端点有没有被遍历到即可。

时间复杂度分析：
顶点数为 n，边数为 e，那么单次深度遍历的时间复杂度为 0（n＋e），由于有 e 条边，那么需要遍历 e 次，所以时间复杂度为 0（en＋e^2），对于稀疏图，e 可以看为 n，所以时间复杂度为 0（n^2），对于稠密图，e 可以看作 n^2，所以时间复杂度为 0（n^4）。但是对于连通块数较多的图，可以有效的减少单次深度遍历的复杂度，算法效率有了提高。

3． 高效算法

3.1 并查集
概念：并查集是一种可以动态维护若干个不重叠的集合，并支持合并与查询两种操作的一种数据结构。合并就是将两个不同的集合合并成一个集合，查询就是查看某个元素属于哪一个集合。

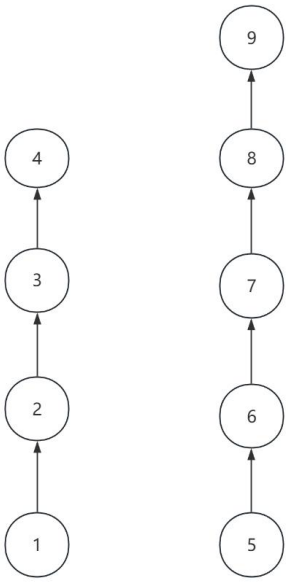

图 4 查询

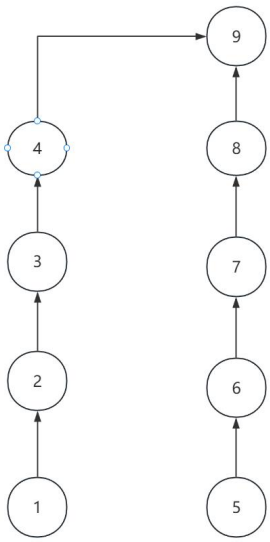

图 5 合并

如图 4，我可以通过父亲节点查询每一个顶点的根节点从而找到它们所属的集合。
如图 5，我只需要设置顶点 4 的父亲节点为 9 就可以把集合（1，2，3，4）和集合（5，6，7，8，9）合并成一个集合。

并查集找桥的思路：
（1）设计一个并查集数组，让每一个顶点都能指向它的根节点。
（2）在创建邻接表的时候动态创建并查集数组，然后求初始情况的连通分量个数。
（3）遍历每一条边，删除该边后生成新的并查集数组，求新的连通分量个数，如果连通分量个数发生改变则该删除的边是桥，反之这个删除的边不是桥。

路径压缩：

对并查集进行路径压缩操作有利于查询的优化，提高查询效率。

如图 4，当我要查询顶点 1 属于哪个集合时，要一层层的递归查找根节点，效率十分低下，我希望把其它的顶点可以放在根顶点下面，每个顶点的父亲顶点都是根顶点，这样可以提高算法的效率。

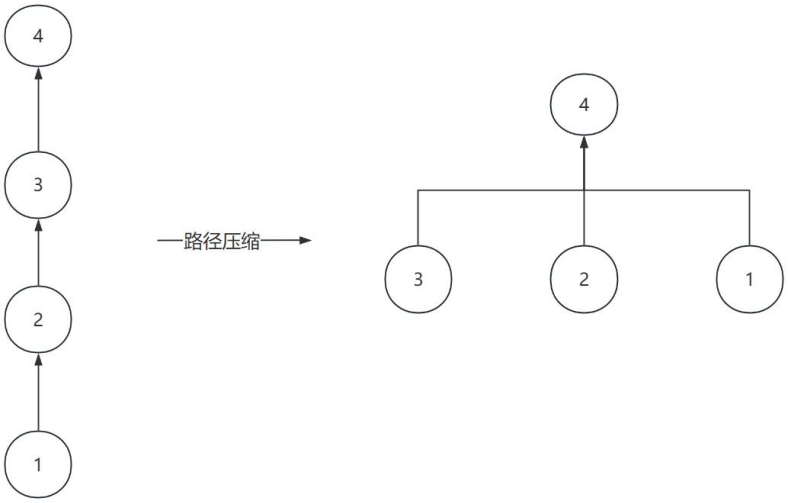

图 6 路径压缩

如图 6，我们可以对并查集进行路径压缩，压缩后我们只需要一次递归就可以找到任意顶点的根顶点。

伪代码：

• 查询：

Algorithm 3：search

Input：index

Output: father_of_index

```
1. If(fa[index] == index)
2.     Return index
3. Else
4.     Return search(fa[index])
```

• 合并：

Algorithm 4：union

Input：index1，index2

Output:none

```
1. Index1_father = search(index1)
2. Index2_father = search(index2)
3. Fa[index1_father] = Index2_father
```

Algorithm 5：search_improved

Input：index1，index2

Output:none

```
1. If(fa[index] == index)
2.    Return index
3. Else
4.      fa[index]=search(fa[index])
5.      Return fa[index]
```

说明：

查询：fa 数组记录着每个顶点的父亲顶点，而根顶点的父亲顶点是自己（递归终止条件），只需要递归就可以找到根顶点。

合并：首先查询两个集合的根顶点，然后把其中一个根顶点的父亲顶点设置为另一个集合的父亲顶点即可。

路径压缩：只需要修改一个查询的代码，在第二个分支中把 father 数组记录的父亲节点都改成根节点。

3.2 并查集＋生成树

生成树：

如果一条边是一座桥，那么它不在任何的环上面，所以对于环上的边，我们不需要进行删除尝试，只需要尝试生成树上的边即可。这样虽然不能完全跳过那些在环上的边，但也在一定程度上排除了不可能的边。

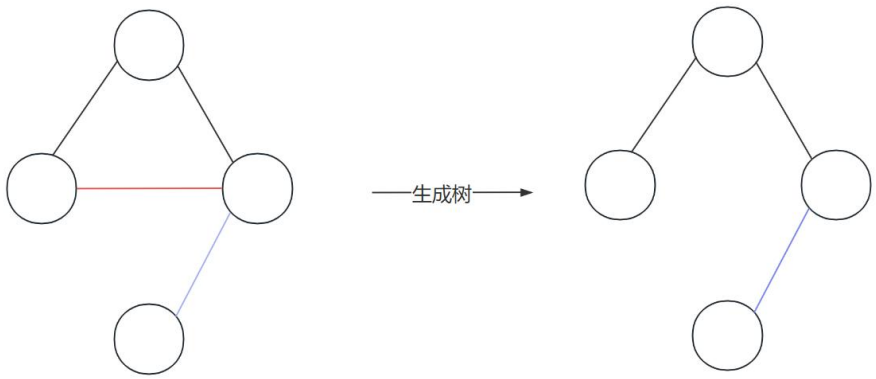

图 7 生成树

如图 7，我生成了左图的生成树，这样遍历的时候就可以减少对红色边的遍历，提高效率，同时蓝色边（桥）也会在被遍历的边中，所以我们可以找到桥。

我使用深度优先遍历来实现生成树。

基本思路：

我先生成原图的 DFS 生成树，之后我只需要列举生成树上的边进行删除验证它是不是桥就可以了。具体就是：

（1）生成原图的生成树

（2）统计原图的连通块数量。

（3）遍历生成树的每一条边，作为删除的边

（4）遍历原图的每一条边，判断它是不是要删除的边，如果不是，则加入图中，使用并查集的合并操作对于这条边连接的两个顶点进行合并。

（5）统计集合的数量，也就是新的连通块的数量。

（6）如果联通块数量增加了，这条边就是桥，反之则不是桥。

伪代码：

Algorithm 6：并查集+生成树

```
Input：none
Output: num_of_bridges
Init ( ) //初始化访问数组和并查集
For i in range (n)://统计原始的连通分量
    If( ! visit [ i ] )
        Blocks_old++;
        DFS( i )
For i in range (n): //求解生成树保存在 Tree_edges 中
    If( ! visit[ i ] )
        DFS_TREE(i , Tree_edges)
For e1 in Tree_edges: //遍历删除生成树上的边
    For e2 in EDGES:
        If( e1 != e2 ):
            Union(e2.index1, e2.index2) //插入所有非删除的边
    Block_new = num_of_set
    If block_new > block_old:
        num_of_bridges++
```

具体的流程解释参考上面的基本思路。

时间复杂度分析：

规定定点数为 n，边数为 e。因此只需要遍历删除生成树上的边，所以删除的次数为 n。每次删除后都需要通过并查集求解连通块个数，所以时间复杂度为 O（e）。因此总体的时间复杂度为 O（ne）。可以推出，对于稀疏图，时间复杂度为 O（n^2），对于稠密图，时间复杂度为 O（n^3）。

3.3 LCA＋并查集

基本思路：

如果一条边它不在环上，那么它就是桥。因此我只需要把所有环上的边找出来，剩

下来的就是桥。

那么要如何找出所有的环上的边呢？

首先，还是需要借助生成树，我同样使用 DFS 来建立生成树，在建立生成树的同时，建立并查集，所以在建立生成树的时候需要一个根顶点的标号 root。此外，还需要记录生成树中每一个顶点的直接前驱。

最近公共祖先（LCA）：

什么是最近最近公共祖先呢，顾名思义就是两个顶点的向上寻找祖先顶点时，第一个出现的顶点，也是两个顶点所有公共祖先顶点里，离根顶点最远的那个节点。

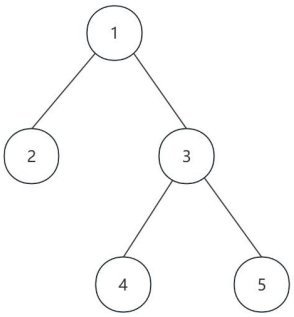

图 8 LCA

如图 8，顶点 4，5 的 LCA 是 3，顶点 2，4 的 LCA 是 1。

那么如何通过 LCA 寻找环上的边呢，当加入非生成树边时，从这条边的两个顶点开始，寻找它们的最近公共祖先，那么寻找公共祖先路径上的所有边都是环边。

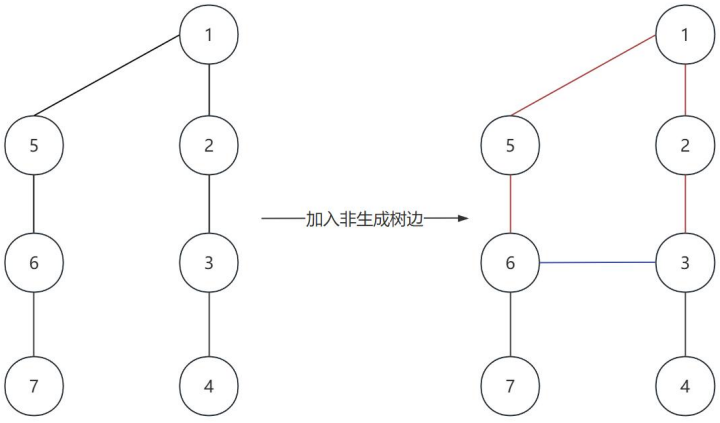

图 9 找环边示例

如上图，当我加入非生成树边（3，6）时，从两个顶点 3，6 寻找最近公共祖先路径上的（5，6），（5，1），（3，2），（2，1）四条边都是环边。

路径压缩：

以图 9 为例，当我们再次添加非生成树边（7，4）时，进行寻找最近公共祖先时，路径分别是 7-6-2-1 和 4-3-2-1，这样会重复查询我们添加边（3，6）时已经找到的

环边，使算法的效率变低，于是需要进行路径压缩。

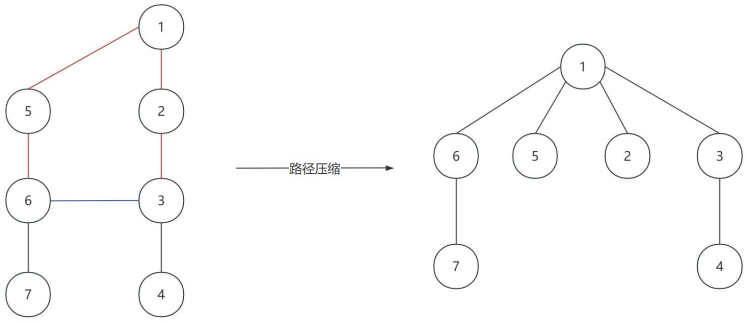

图 10 路径压缩

当我添加边（3，6）进行寻找最近公共祖先后，对于路径上的所有顶点，改变它们的父亲顶点为根顶点。

伪代码：

Algorithm 7：并查集+LCA

Input：none

Output: num_of_bridges

```
1.  For e in Edge:
2.      If e in Tree_edges: continue
3.      d1=deep[v1]    d2=deep[v2] //v1,v2 是 e 的两个顶点
4.  If( d1 > d2 ):
5.        While(d1>d2):
6.              Visit[v1]=1 father[v1]=union[v1] deep[v1]=1
7.              V1=father[v1]      d1=deep[v1]
8.  Else if( d1 < d2 ):
9.        While(d1<d2):
10.             Visit[v2]=1 father[v2]=union[v2] deep[v2]=1
11.             V2=father[v2]      d1=deep[v2]
12.  While( v1 != v2 ):
13.       Visit[v1]=visit[v2]=1
14.       father[v1]=union[v1]    father[v2]=union[v2]
15.       deep[v1]=1    deep[v2]=1
16.       V1 =father[v1]    v2=father[v2]
17.  Num_of_bridges = NODES-count（visit[]==1）
```

说明：

对于图上的所有边，先判断它是不是生成树上的边，如果是则直接跳过，如果不是生成树上的边，那么就找出这条边的两个端点，如果它们不处于同一深度，则先让深度大的寻找父亲节点使它们深度一样，之后再一起遍历寻找最近公共祖先，寻找路径上的点都标记为 1，代表它们与父亲连接的边是环边不是桥。最后只需要使用顶点的数量加上集合的数量再减去环边数量即可得到桥的数量。

时间复杂度分析：

n 为顶点数量，e 为边的数量，遍历所有边集的时间复杂度为 O（e）。一次查找公共祖先的最坏情况下时间复杂度是 O（n），但是我使用了路径压缩算法，所以树高基本是一个常数，因此可以优化时间复杂度为 O（1），加上建树的时间复杂度，总体的时间复杂度为 O（n＋e）。对于稀疏图，时间复杂度为 O（n），对于稠密图，时间复杂度为 O（n^2）。

4. 算法测试

正确性测试：

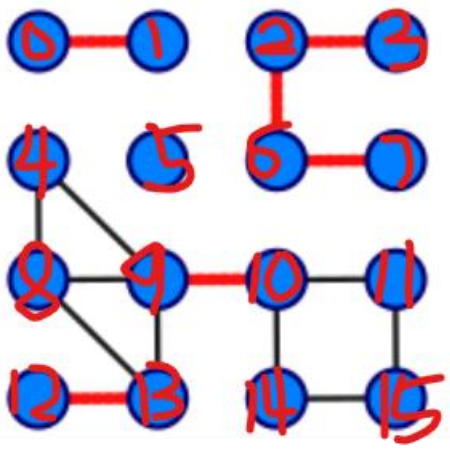

图 11 标记后的图 2

对图 2 进行标记，如图 11 所示，之后我就可以使用图 11 测试我的算法的正确性了。

• 基准算法（通过连通块数量）：

表一：基准算法（通过连通块数量）测试结果

桥的数量	桥
6	（0，1） （2，3） （2，6） （6，7） （9，10） （12，13）

从表一的结果我们可以得出结论我们的基准算法（通过连通块数量）是正确的。

• 基准算法（连通性）：

表二：基准算法（连通性）测试结果

桥的数量	桥
6	（0，1） （2，3） （2，6） （6，7） （9，10） （12，13）

从表二的结果我们可以得出结论我们的基准算法（连通性）是正确的。

• 并查集+生成树：

表三：并查集+生成树测试结果

桥的数量	桥
6	（1，0） （3，2） （6，2） （7，6） （10，9） （12，13）

从表三的结果我们可以得出结论我们的并查集+生成树算法是正确的。

• 并查集+LCA：

表四：并查集+LCA 测试结果

桥的数量	桥
6	（1，0） （3，2） （6，2） （7，6） （10，9） （12，13）

从表四的结果我们可以得出结论我们的并查集+LCA 算法是正确的。

算法性能测试：

表五：算法性能测试

算法	基准算法 (连通块)	基准算法 （连通性）	并查集 +生成树	并查集 +LCA
mediumDG	0.430ms	0.399ms	0.324ms	0.0043ms
largeG	-	-	-	0.89s

mediumDG 有 50 个顶点，147 条边，0 座桥，因为整幅图只有一个连通子图，所以两种基准算法并没有太大的差异。使用并查集＋生成树的算法也并没有达到数量级上的时间优化。最高效的就是并查集＋LCA 算法，它对于稀疏图有着线性的时间复杂度。

largeG 有 1000000 个顶点，7586063 条边，8 座桥，分别是（372243，148837），（467595，907820），（548437，461822），（589095，317390），（630627，467595），（639238，969090），（658123，724640），（773903，95760）。除了并查集＋LCA 算法，其它算法都无法在 largeG 上跑出结果。

随机数据测试效率：

在这一步骤中，我对于基准算法和最高效的算法并查集＋LCA 测试在稀疏图和稠密图中的算法性能。

稀疏图中的测试：

表六：稀疏图中的测试（单位为 ms，边数和顶点数一样）

规模	1000	1500	2000	2500
基准算法	27	54	98	163
并查集＋LCA	0.12	0.16	0.21	0.32

可以看出高效算法的速度远远大于基准算法。

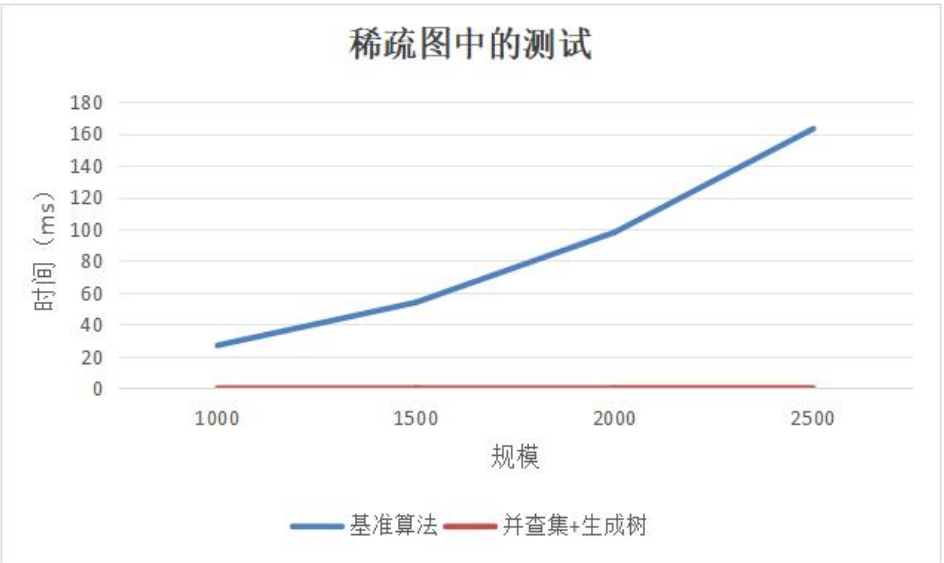

图 12 稀疏图下的性能比较

通过曲线图对比发现两个算法的时间根本不是一个数量级的。

稠密图中的测试：

我将边的数量设置为顶点数量的平方除以 4

表七：稠密图中的测试（单位为 ms）

规模	n=100	N=200	N=300	N=400
基准算法	65	985	4631	14531
并查集＋LCA	0.06	0.21	0.43	0.87

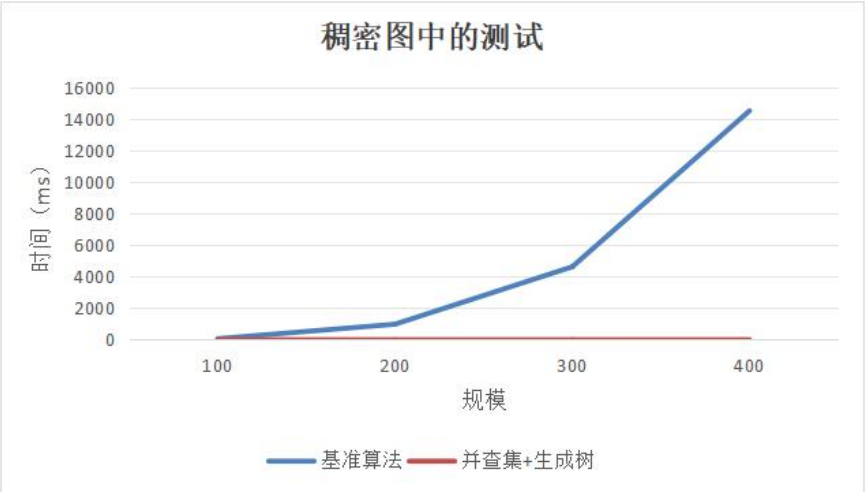

图 13 稠密图下的性能比较

在稠密图中，高效算法的优势就更加明显了。

五、实验结果及分析：

本次实验，我先实现了基于连通块数的基准算法，之后我又对基准算法进行了改进，使用了基于连通性的基准算法。然后，我开始寻求基于并查集的高效算法，我先是使用并查集和生成树的算法，这使时间复杂度有了一定的优化，但并没有在大规模数据下有很好的表现，于是我继续改进，我使用了 LCA 和并查集算法，这成功的在稀疏图上达到了线性的复杂度，在大规模数据上有着不错的表现。

我还随机生成了不同规模的稀疏图数据和稠密图数据检测基准法和高效算法的性能。发现，高效算法相比基准算法确实是有着数量级的提升，而且在稠密图中这种提升会更加明显，这是因为高效算法的时间复杂度分别优化到了 O（n）和 O（n^2），但是基准算法的时间复杂度是 O（n^2）和 O（n^4）。

总体上，我实现了基准算法并构建了高效的算法，在提供的数据集和随机生成的数据集上都取得了很好的效果，实验圆满完成。

六、实验结论：

1. 对于稀疏图，使用邻接表进行存储会有更高的效率。

2. 使用基准算法在稀疏图上的时间复杂度为 O（n^2），在稠密图上的时间复杂度为 O（n^4）。使用连通性对基准算法进行改变并没有优化时间复杂度。不过在连通分支更多的图上会有更好的表现。

3. 使用并查集＋生成树的算法在稀疏图上的时间复杂度为 O（n^2），在稠密图上的时间复杂度为 O（n^3）。

4. 使用并查集＋LCA 的算法在稀疏图上的时间复杂度为 O（n），在稠密图上的时间复杂度为 O（n^2），是最高效的算法。

5. mediumDG 上没有桥，四种算法都可以跑出结果，largeG 上有八座桥，但是只有 LCA＋并查集这个算法可以跑出结果，其余三个都不能在可接受的时间内跑出结果，这也体现了这个算法的高效性。在不同的算法中，找到桥的顺序也是不一样的。

6. 在随机生成的数据上对比基准算法和高效算法的时间，更体现了高效算法的优越性。其中在稠密图中的性能差异会更加的明显。

7. 设计图的算法时，要重视桥的定义与特性，桥不在任何环上这个性质是我设计高效算法的关键。

8. 设计图算法的同时，可以考虑使用树来优化，树是一种特殊的图，最高效的算法也是需要用到生成树的。

指导教师批阅意见：   **成绩评定：**   **指导教师签字：** **2024 年 6 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。