**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）**

**学院： 计算机与软件学院 专业：计算机科学与技术（创新班）**

**报告人： XXX 学号： XXXXXXXXXX 班级： 高性能班**

**同组人：**

**指导教师： 杨烜**

**实验时间： 2024.6.2——2024.6.7**

**实验报告提交时间： 2024.6.7**

**教务处制**

**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、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。

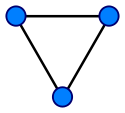
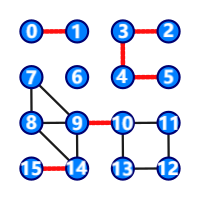
 

图1没有桥的无向连通图 图2这是有16个顶点和6个桥的图

（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。

**三、内容**

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。

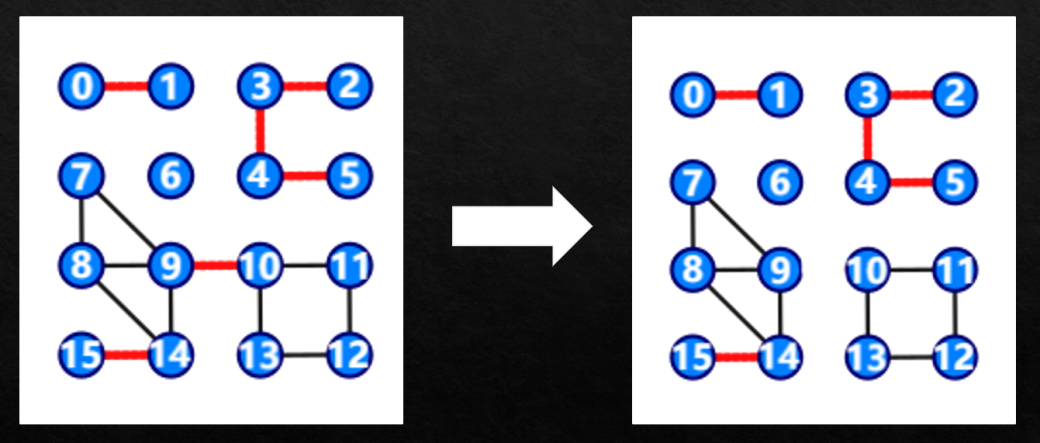
**四、求解问题的算法原理描述**

**（一）基准算法**

1、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。

例如在下图中，节点9和节点10之间的边e为桥，所以当边e从图中删去后，原本图中的连通块个数从4个变成了5个。



利用以上原理，我们可以得到以下算法：

* 先用DFS统计图的连通分量个数num1。
* 接着遍历边集中的每条边e，在图中删除该被遍历的边e。
* 然后再次用DFS获取整张图的连通分量个数num2。
* 如果num2>num1，说明边e为桥；如果num1>num2，则说明边e不是桥。
* 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。

而DFS统计图的连通分量的方法为：

对每个顶点执行DFS，如果顶点尚未被访问，则从该顶点开始DFS遍历，标记所有与该顶点连通的顶点，并将其标记为同一个连通块。每次开始新的DFS遍历时，连通块计数器加一，表示发现了一个新的连通块。遍历结束后，连通块计数器的值即为图中连通块的总数。

1. 复杂度分析

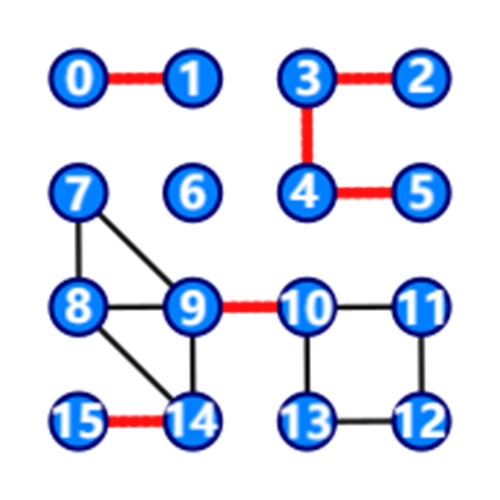
顶点个数为n，边数为e，使用邻接表的情况下，一次DFS遍历图的时间复杂度为。由于需要遍历每一条边，则要进行e次DFS操作，所以时间复杂度为 。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为。

1. 正确性验证

以下图为输入，验证算法是否正确。



输出结果如下图，算法的正确性得以验证。



1. **基准算法+判断相连**

1、算法思想

可以注意到，在上面的算法中，由于需要统计连通块个数变化来判断边是否为桥，所以DFS需要搜索所有的连通块，因此会导致算法的效率较低。

实际上，如果删除的边e为桥，则只要判断删除边e后边的两端顶点u和v是否还相连即可，这是因为存在边e，就意味着顶点u和v初始时是在同一个连通块中的。根据桥的定义，如果边e不为桥，则不会导致u和v所在的连通块分裂成两块，因此由于u和v还在同一个连通块中，顶点u和v之间仍保持互通（即顶点u到v之间仍存在路径可达）；如果边e为桥，则会导致u和v所在的连通块分裂成两块，并且u和v各在一个不同的连通块上，如此顶点u和v之间则不互通。

根据以上分析，可以得到以下算法：

* 遍历边集中的每条边e，在图中删除该被遍历的边e，记录下边e两端顶点u和v。
* 然后用DFS查看顶点u和v之间是否互通。
* 如果不互通，说明边e为桥；如果互通，则说明边e不是桥。
* 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。

通过DFS查看图中两个顶点是否互通的方法：

从一个顶点开始执行DFS遍历，标记所有访问过的顶点。如果在遍历过程中访问到了目标顶点，则这两个顶点是连通的；否则，遍历结束后未访问到目标顶点，则这两个顶点不连通。

通过以上方法，就可以避免花费大量时间统计全部连通块的数量，只需要在被删除边所在的连通块上进行DFS遍历即可。

2、复杂度分析

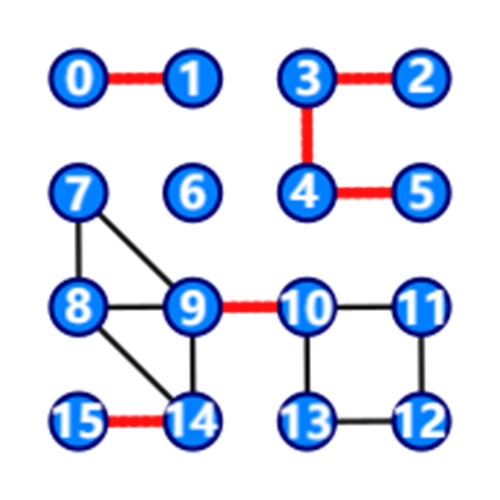
顶点个数为n，边数为e，使用邻接表的情况下，一次DFS遍历图的时间复杂度为。由于需要遍历每一条边，则要进行e次DFS操作，所以时间复杂度为 。

对于稀疏图， ，所以时间复杂度为。

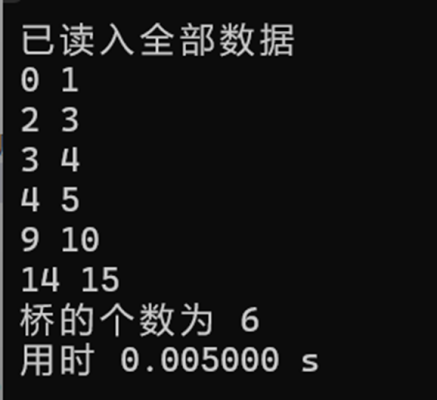
对于稠密图， ，所以时间复杂度为。

3、正确性验证

以下图为输入，验证算法是否正确。



输出结果如下图，算法的正确性得以验证。



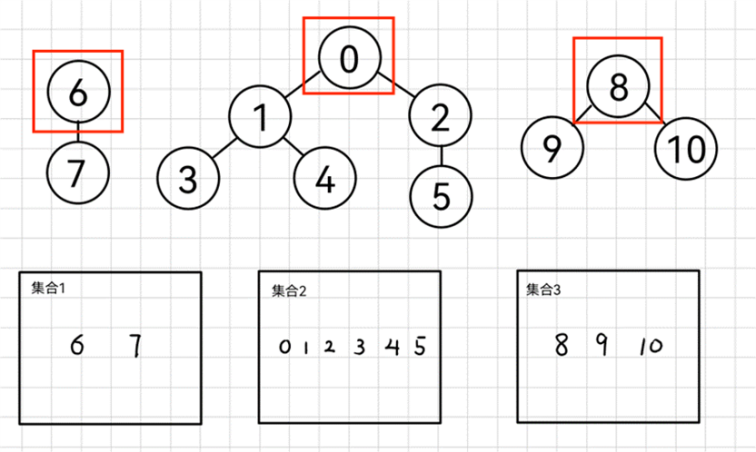
1. **基准算法+并查集**

1、算法思想

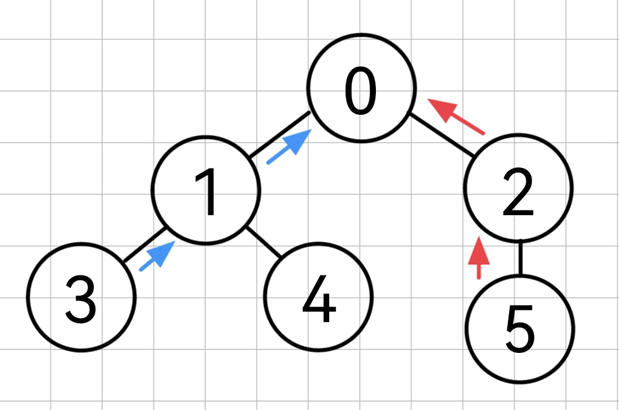
并查集是一种树型的数据结构，用于处理集合合并和查询的问题。

在并查集中，每个集合为一颗树，树上的节点为该集合中的元素，树的根节点表示该集合的代表元素。如果需要寻找集合的代表元素，只需要一层一层往上访问父节点，直达树的根节点即可得到代表元素，因此，判断两个元素是否在同一个集合中，只需要向上探查这两个元素的父亲元素，直到得到代表元素，若代表元素相同，则说明连个元素位于同一个集合中。通过该方法，可以处理查询两元素是否位于同一集合的问题。

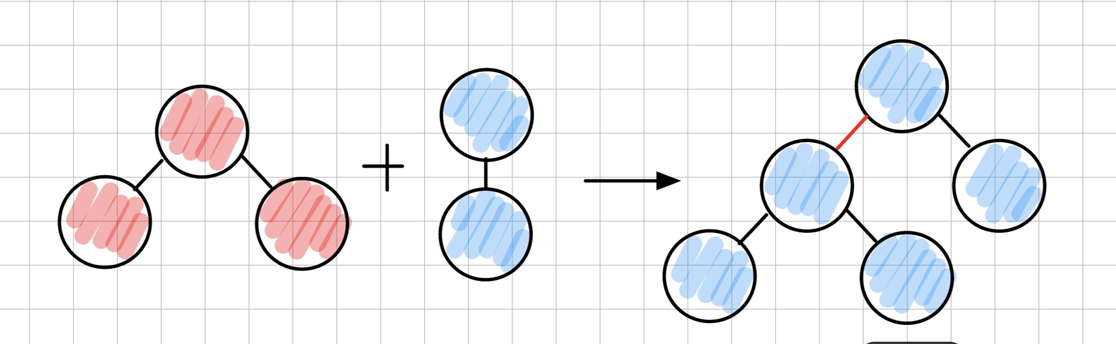
如下图，集合2中有元素0、1、2、3、4、5，集合2的代表元素为0，因此形成了以元素0为根节点、节点数为6的树，可以注意到，位于该树上的所有节点，向上探查的终点皆为元素0。



下图则展示了寻找代表元素的过程，如元素3，其向上探查父亲节点得到元素1，元素1再向上探查父亲节点得到元素0，由于元素0为根节点，因此无法再向上探查，故元素0即为元素3的代表元素。



需要合并两个集合，只需要将一个集合的代表元素的父亲节点改为另一个集合的代表元素。如下图，红色节点和蓝色节点分别代表集合1和集合2，同时它们形成了两颗不同的树，若要将集合1合并入集合2，则只需要将集合1的代表元素，即集合1根节点的父亲节点改为集合2的代表元素，即集合2根节点。如此，集合1中全部元素向上探查得到的代表元素即为集合2的代表元素，集合1成功合并入集合2。



根据以上并查集的原理，很容易联想到在图中应用并查集，顶点代表元素，顶点之间的边表示两个顶点属于同一个集合，不同的连通块对应于并查集中不同的集合。通过图中的边可以逐步合并顶点所在的集合，最终构建出多个不同的集合，每个集合代表一个连通块，这些集合的个数则表示图中的连通块数量。

根据以上分析，可以得到以下算法：

* 先用并查集统计图的连通分量个数num1。
* 接着遍历边集中的每条边e，在图中删除该被遍历的边e。
* 然后再次用并查集获取整张图的连通分量个数num2。
* 如果num2>num1，说明边e为桥；如果num1>num2，则说明边e不是桥。
* 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。

通过并查集统计图中连通块个数的方法：

初始时每个顶点各自为一个集合，根据图中的边进行合并操作，将边两端的顶点合并到同一个集合。遍历所有边后，集合数量即为图中连通块的个数。

2、复杂度分析

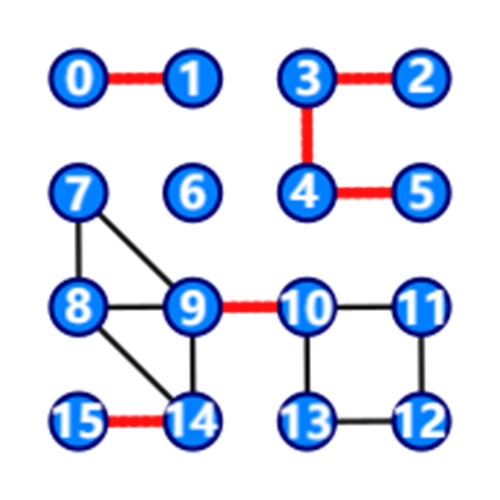
顶点个数为n，边数为e，最坏情况下合并两个集合的时间复杂度为。每次计算图中集合个数，初始化的时间复杂度为，合并集合需要遍历e条边，所以时间复杂度为。由于需要遍历每一条边，则有e次计算图中集合个数的过程，所以时间复杂度为。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为。

3、正确性验证

以下图为输入，验证算法是否正确。



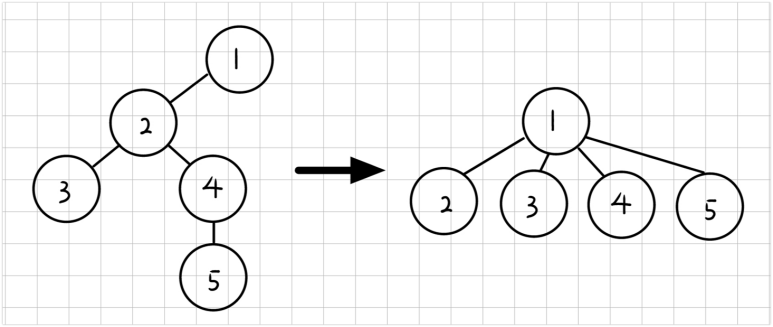
输出结果如下图，算法的正确性得以验证。



1. **基准算法+并查集+路径压缩**
2. 算法思想

在并查集中，存在路径压缩的操作，即在查找某个元素的代表元素，即根节点时，将沿途访问的所有节点直接连接到根节点上，从而扁平化树结构，减少树的高度。这种优化在后续查找操作中显著加快了访问速度，提高了并查集的整体性能。

如下图，原本元素5查找代表元素需要向上探查3次，如果路径压缩后，则只需要向上探查1次即可得到代表元素。



根据以上分析，可以得到以下算法：

* 先用并查集统计图的连通分量个数num1。
* 接着遍历边集中的每条边e，在图中删除该被遍历的边e。
* 然后再次用并查集获取整张图的连通分量个数num2。
* 如果num2>num1，说明边e为桥；如果num1>num2，则说明边e不是桥。
* 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。

通过并查集统计图中连通块个数的方法：

初始时每个顶点各自为一个集合，根据图中的边进行合并操作，将边两端的顶点合并到同一个集合，通过路径压缩优化合并过程。遍历所有边后，集合数量即为图中连通块的个数。

2、复杂度分析

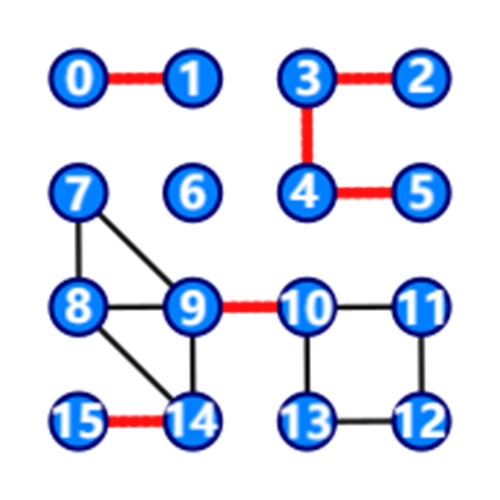
顶点个数为n，边数为e，路径压缩后合并两个集合的时间复杂度为（根据网上查阅资料可知，使用路径压缩后，并查集的每个操作平均时间仅为 ，其中为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数）。每次计算图中集合个数，初始化的时间复杂度为，合并集合需要遍历e条边，所以时间复杂度为。由于需要遍历每一条边，则有e次计算图中集合个数的过程，所以时间复杂度为。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为。

3、正确性验证

以下图为输入，验证算法是否正确。



输出结果如下图，算法的正确性得以验证。



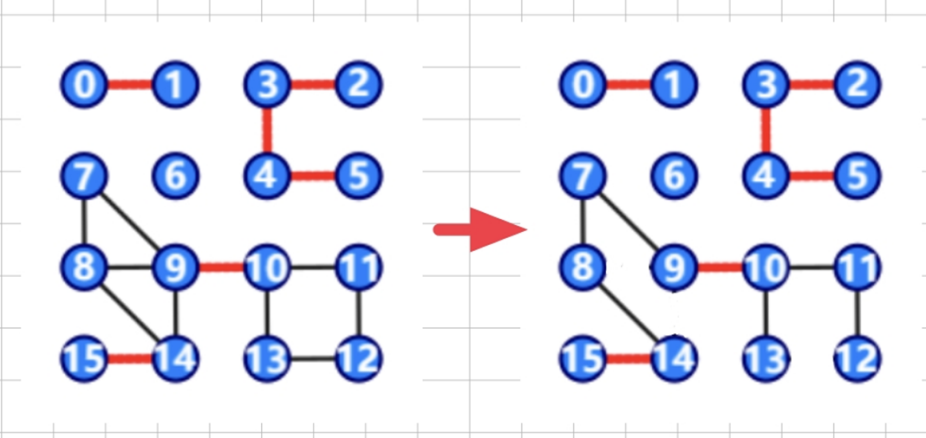
1. **生成树+LCA**

1、算法思想

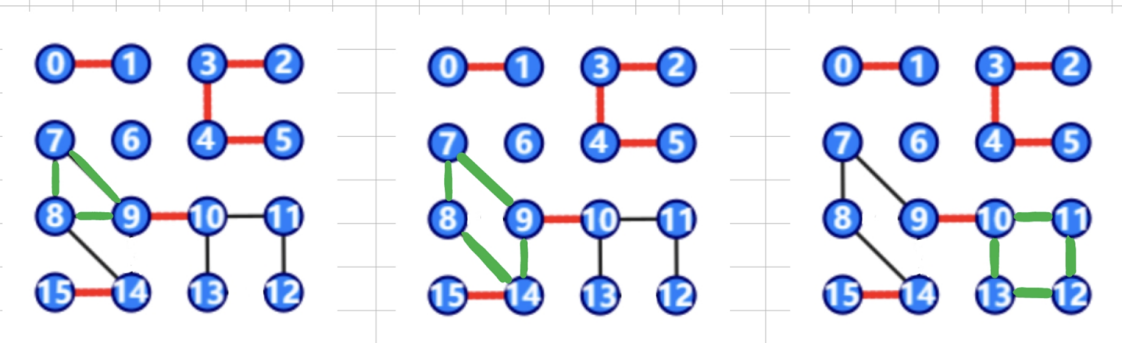
根据桥的定义，“一条边是一座桥当且仅当这条边不在任何环上”，故桥总数=边总数 - 环边总数，问题就可以转化为了求图中的环边数量。

首先，根据图论的知识可以知道，树是边数最小的无环图，在树上任意两顶点间加边，则会导致图中形成环。

在下图中，左边的图为原本的图G，右边的图为图G的生成树，可以发现，顶点8和顶点9、顶点9和顶点14、顶点12和顶点13之间边不在生成树上。



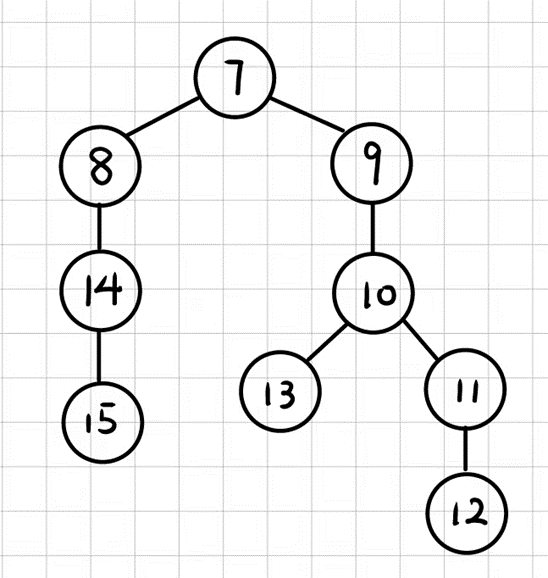
如果将顶点8和顶点9、顶点9和顶点14、顶点12和顶点13之间边重现添加到生成树上，可以发现，均会在图中形成一个环（即绿色线段围起的环）。



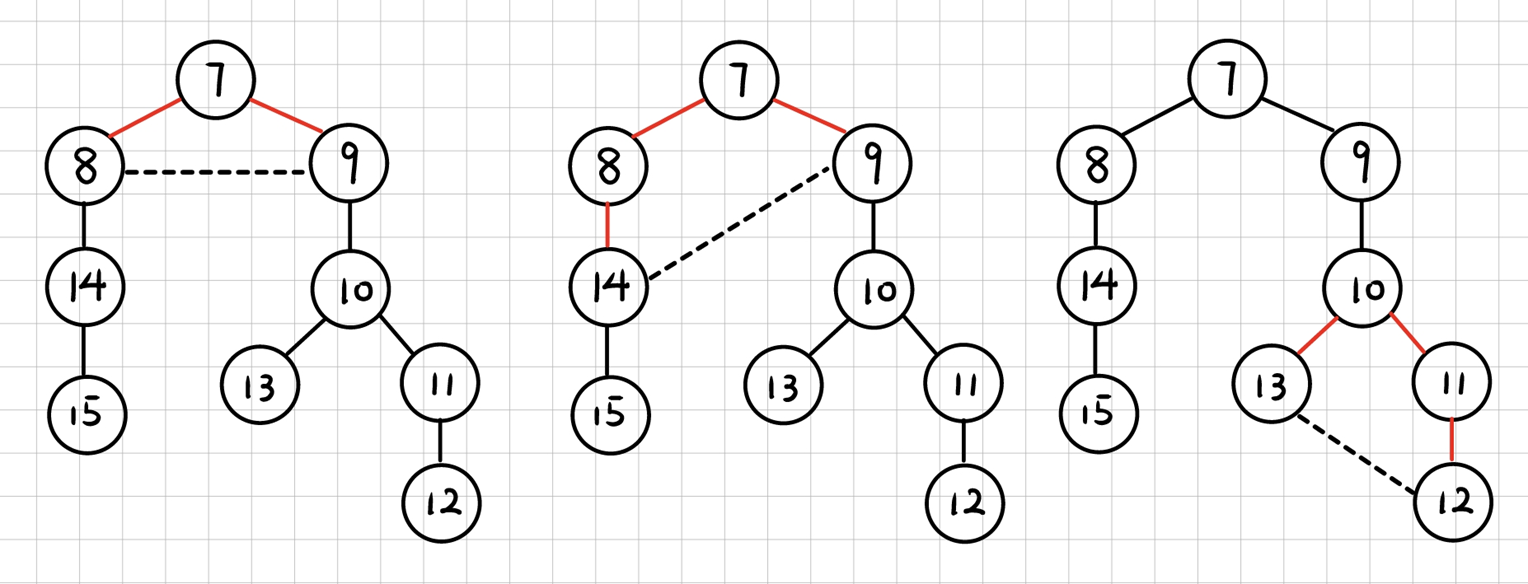
因此，我们可以确认，图中的某一条边若不在该图的生成树上，则该边一定为环边。

此外，将非生成树上的边添加到生成树上，可以发现对于这条边的两个节点，向上寻找它们最近公共祖先节点（LCA）路径上所有经过的边也均为环边。

如下图为一个图的生成树



假设顶点14和顶点9之间存在非生成树上的边e，将边e添加到生成树上，可以发现顶点14和顶点9的公共祖先顶点7与这两个顶点互通的路径上经过的边均为环边。



根据以上分析，可以得到以下算法：

* 通过DFS/BFS获得图的生成树。
* 统计非生成树边的个数，将它们记录为环边。
* 将非生成树边加入生成树之中，将形成的环上未标记的环边（即非生成树边的两端节点寻找LCA的路径上经过的边）进行记录。
* 重复第二个步骤，直到所有非生成树边都已经被遍历。
* 最终桥的个数=边总数-环边个数

2、复杂度分析

不同建树方法有不同的时间复杂度。

用DFS生成树：

顶点个数为n，边数为e，DFS生成树的时间复杂度为。依次向生成树加入非生成树边，由于DFS建树的深度较深，每次查找LCA，最坏情况下该过程时间复杂度为，所以统计树上环边的时间复杂度为。整个算法的时间复杂度为。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为。

用BFS生成树：

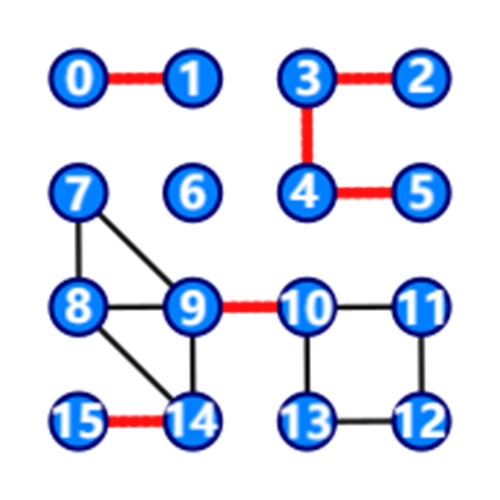
顶点个数为n，边数为e，BFS生成树的时间复杂度为。依次向生成树加入非生成树边，由于BFS建树的深度较浅，每次查找LCA，该过程平均时间复杂度为，所以统计树上环边的时间复杂度为。整个算法的时间复杂度为。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为

3、正确性验证

以下图为输入，验证算法是否正确。



输出结果如下图，算法的正确性得以验证。



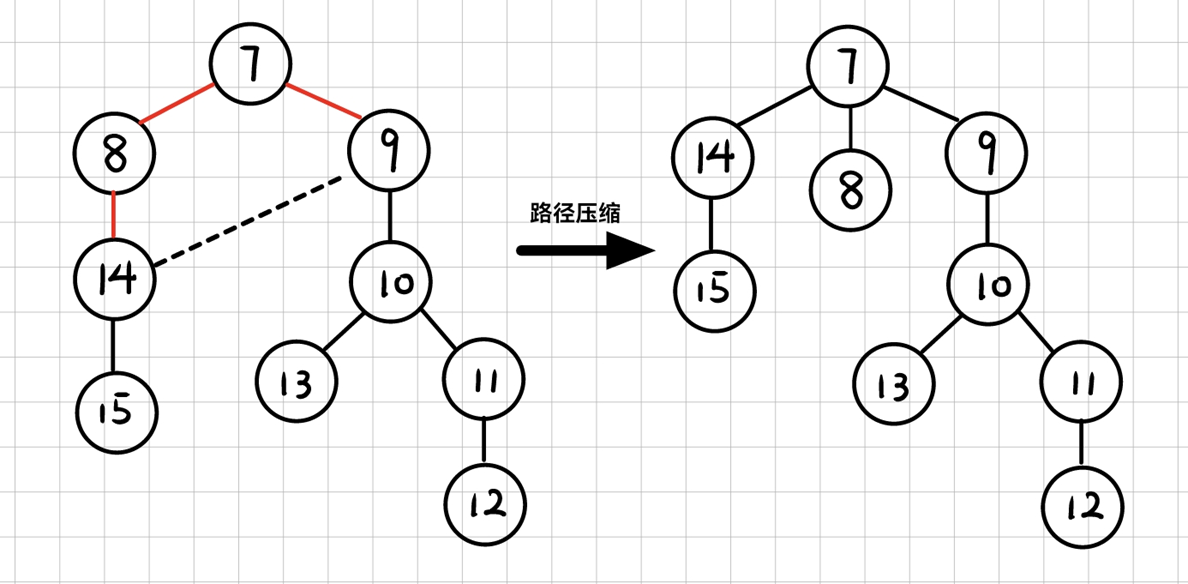
1. **生成树+LCA+路径压缩**

1、算法思想

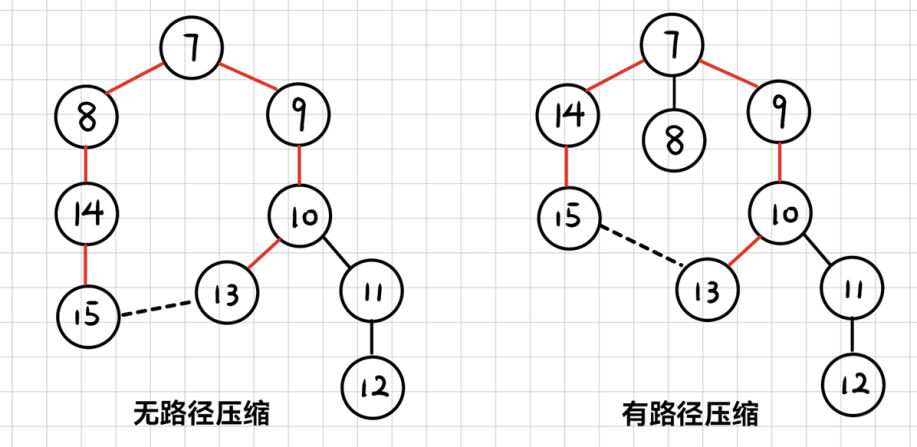
可以发现在上面生成树+LCA的算法中，查找两个顶点的LCA会花费大量的时间。

结合我们前面对并查集的分析，可以想到用并查集中的路径压缩来缩短查找两个顶点LCA的时间。具体而言，便是当向上查找LCA过程中经过路径被标记后，后续再遍历不会对答案有任何贡献，如此不如直接将该路径压缩，减少下次向上查找经过该路径所需要的时间。我们可以在标记环边后，将已标记环边的儿子节点的父亲节点改为向上查找得到的LCA，如此在下次向上查找时，则可以直接跳到该儿子节点的LCA，减少时间花销，从而达到压缩路径的目的。

如下图，在查找得到节点9和节点14的LCA节点7后，可以进行路径压缩，即将查找LCA路径上经过的节点8、节点9、节点14的父亲节点改为LCA节点7（节点8和节点9的父亲节点已经为节点7则不需要修改）。



路径压缩后，假设需要查找节点13和节点15的LCA，向上探查LCA的总次数为5次，相比于没有路径压缩的6次，可以发现向上探查LCA确实由于路径压缩而减少。可以想到，如果生成树越深，路径压缩则更有利于减少二次探查所花费的时间。



根据以上分析，可以得到以下算法：

* 通过DFS/BFS获得图的生成树。
* 统计非生成树边的个数，将它们记录为环边。
* 将非生成树边加入生成树之中，将形成的环上未标记的环边（即非生成树边的两端节点寻找LCA的路径上经过的边）进行记录。
* 记录下LCA，从非生成树边的两端节点开始重新向上探查，探查过程中将所有经过的节点的父亲节点改为LCA，直到经过LCA停止探查。
* 重复第二个步骤，直到所有非生成树边都已经被遍历。
* 最终桥的个数=边总数-环边个数

2、复杂度分析

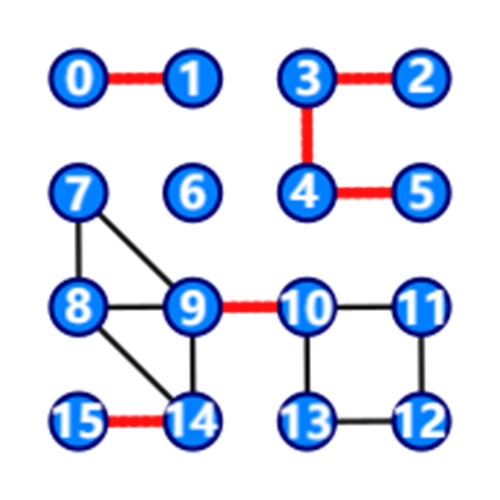
顶点个数为n，边数为e，DFS/BFS生成树的时间复杂度为。依次向生成树加入非生成树边，每次查找LCA，该过程时间复杂度为（类似于并查集的路径压缩操作，使用路径压缩后，每次查找LCA的操作平均时间仅为 ，其中为阿克曼函数的反函数，其增长极其缓慢，也就是说其单次操作的平均运行时间可以认为是一个很小的常数），所以统计树上环边的时间复杂度为。整个算法的时间复杂度为。

对于稀疏图， ，所以时间复杂度为。

对于稠密图， ，所以时间复杂度为。

3、正确性验证

以下图为输入，验证算法是否正确。



输出结果如下图，算法的正确性得以验证。



1. **各算法伪代码**
2. **基准算法**

|  |
| --- |
| function Basic\_Algorithm(graph)  num1 <— Count\_Blocks(graph)  bridge\_num <— 0  for edge in graph  Remove\_From\_Graph(edge)  num2 <— Count\_Blocks(graph)  if num2 > num1  bridge\_num <— bridge\_num + 1 // 如果连通块个数变化，则桥数加一  end if  end for  return bridge\_num  end function  // 统计连通块个数  function Count\_Blocks(graph)  cnt <— 0 // 连通块个数  for vertex in graph  if visited[vertex] != 1  DFS(vertex) // 从vertex开始遍历图  visited[vertex] = 1  cnt <— cnt + 1  end if  end for  return cnt  end function |

1. **基准算法+判断相连**

|  |
| --- |
| function Basic\_Algorithm\_Upgrade(graph)  bridge\_num <— 0  for edge in graph  Remove\_From\_Graph(edge)  check <— Check\_Connection(edge.u, edge.v, edge.v) // 检查edge两个顶点是否相连  if check = 0  bridge\_num <— bridge\_num + 1 // 如果edge两个顶点不相连，则桥数加一  end if  end for  return bridge\_num  end function  // 检查两个顶点是否相邻  function Check\_Connection(now\_vertex, last\_vertex, end\_vertex)  if now\_vertex = end\_vertex  return 1  end if  for edge\_connect\_with\_now\_vertex in graph  if visited[edge\_connect\_with\_now\_vertex] != 1  visited[edge\_connect\_with\_now\_vertex] = 1  u <— edge\_connect\_with\_now\_vertex.v  return Check\_Connection(u, now\_vertex, end\_vertex)  end if  end for  return 0  end function |

1. **基准算法+并查集**

|  |
| --- |
| function Basic\_Algorithm\_with\_DisjointSet(graph)  num1 <— Count\_Blocks\_with\_DisjointSet(graph)  bridge\_num <— 0  for edge in graph  Remove\_From\_Graph(edge)  num2 <— Count\_Blocks\_with\_DisjointSet(graph)  if num2 > num1  bridge\_num <— bridge\_num + 1 // 如果连通块个数变化，则桥数加一  end if  end for  return bridge\_num  end function  // 并查集统计连通块个数  function Count\_Blocks\_with\_DisjointSet(graph)  Init() // 初始化并查集  blocks\_num <— n // 初始化连通块个数为节点数  for edge in graph  fa <— find(edge.u)  fb <— find(edge.v)  if fa != fb  f[fa] <— fb  blocks\_num <— blocks\_num - 1 // 连通块个数减一  end if  end for  return blocks\_num  end function  // 查询顶点x的代表顶点  function find(x)  if f[x] = x  return x  else  return find(f[x])  end if  end function |

1. **基准算法+并查集+路径压缩**

|  |
| --- |
| function Basic\_Algorithm\_with\_DisjointSet(graph)  num1 <— Count\_Blocks\_with\_DisjointSet(graph)  bridge\_num <— 0  for edge in graph  Remove\_From\_Graph(edge)  num2 <— Count\_Blocks\_with\_DisjointSet(graph)  if num2 > num1  bridge\_num <— bridge\_num + 1 // 如果连通块个数变化，则桥数加一  end if  end for  return bridge\_num  end function  // 并查集统计连通块个数  function Count\_Blocks\_with\_DisjointSet(graph)  Init() // 初始化并查集  blocks\_num <— n // 初始化连通块个数为节点数  for edge in graph  fa <— find(edge.u)  fb <— find(edge.v)  if fa != fb  f[fa] <— fb  blocks\_num <— blocks\_num - 1 // 连通块个数减一  end if  end for  return blocks\_num  end function  // 查询顶点x的代表顶点  function find(x)  if f[x] = x  return x  else  return f[x] <— find(f[x])  end if  end function |

1. **生成树+LCA**

|  |
| --- |
| function Basic\_Algorithm\_with\_LCA(graph)  BFS() // bfs生成树  edge\_not\_on\_tree\_num <— 0 // 统计不在生成树上的边  for edge\_not\_on\_tree in graph  Find\_LCA(edge\_not\_on\_tree.u, edge\_not\_on\_tree.v) // 查找最近祖先  edge\_not\_on\_tree\_num <— edge\_not\_on\_tree\_num + 1  end for  return m - edge\_not\_on\_tree\_num - loop\_edge\_num  end function  // 查找最近祖先，并标记环边  function Find\_LCA(u, v)  while 1  if depth[u] > depth[v]  u = father[u]  loop\_edge\_num <— loop\_edge\_num + 1  else if depth[u] < depth[v]  v = father[v]  loop\_edge\_num <— loop\_edge\_num + 1  else if u != v  u = father[u]  v = father[v]  loop\_edge\_num <— loop\_edge\_num + 2  else  break  end if  end while  end function |

**（六）生成树+LCA+路径压缩**

|  |
| --- |
| function Basic\_Algorithm\_with\_LCA(graph)  BFS() // bfs生成树  edge\_not\_on\_tree\_num <— 0 // 统计不在生成树上的边  for edge\_not\_on\_tree in graph  Find\_LCA(edge\_not\_on\_tree.u, edge\_not\_on\_tree.v) // 查找最近祖先  edge\_not\_on\_tree\_num <— edge\_not\_on\_tree\_num + 1  end for  return m - edge\_not\_on\_tree\_num - loop\_edge\_num  end function  // 查找最近祖先，并标记环边  function Find\_LCA(u, v)  while 1  if depth[u] > depth[v]  u = father[u]  loop\_edge\_num <— loop\_edge\_num + 1  else if depth[u] < depth[v]  v = father[v]  loop\_edge\_num <— loop\_edge\_num + 1  else if u != v  u = father[u]  v = father[v]  loop\_edge\_num <— loop\_edge\_num + 2  else  break  end if  end while  Zip\_Way() // 压缩路径  end function |

**六、算法测试结果及效率分析**

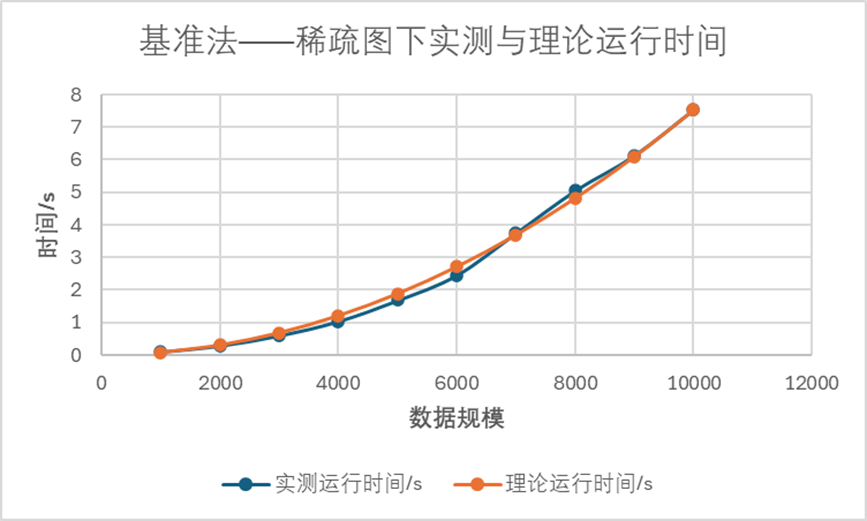
1. **基准算法**

对于稀疏图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.102100 | 0.075267 |
| 2000 | 0.291200 | 0.301068 |
| 3000 | 0.599700 | 0.677403 |
| 4000 | 1.040800 | 1.204272 |
| 5000 | 1.681200 | 1.881675 |
| 6000 | 2.452100 | 2.709612 |
| 7000 | 3.741900 | 3.688083 |
| 8000 | 5.046700 | 4.817088 |
| 9000 | 6.124700 | 6.096627 |
| 10000 | 7.526700 | 7.526700 |



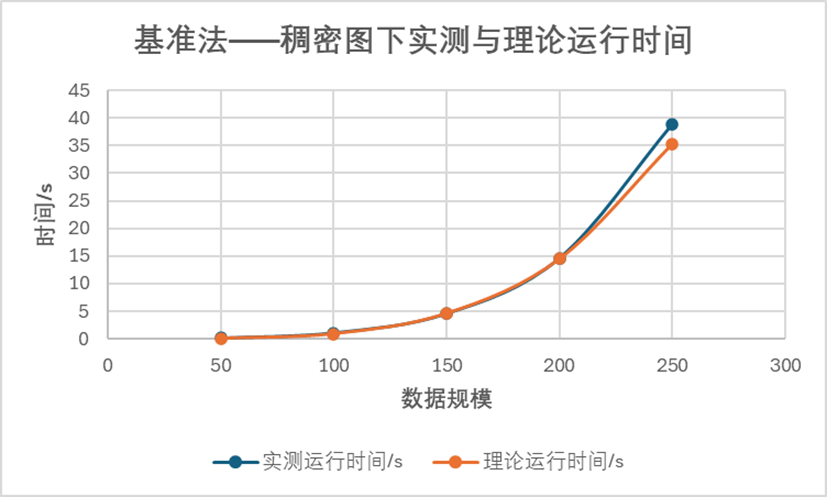
可以看到基本拟合，说明在稀疏图中基准法符合的时间复杂度。

对于稠密图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 50 | 0.122200 | 0.056453 |
| 100 | 1.085000 | 0.903249 |
| 150 | 4.572700 | 4.572700 |
| 200 | 14.596300 | 14.451990 |
| 250 | 38.841600 | 35.283179 |



可以看到基本拟合，说明在稠密图中基准法符合的时间复杂度。

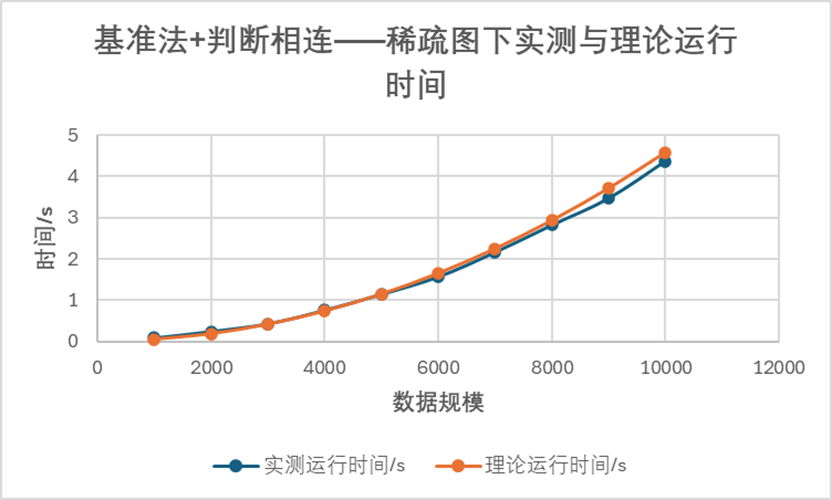
**（二）基准算法+判断相连**

对于稀疏图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法+判断相连——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.084000 | 0.045768 |
| 2000 | 0.240200 | 0.183072 |
| 3000 | 0.428200 | 0.411912 |
| 4000 | 0.756000 | 0.732288 |
| 5000 | 1.144200 | 1.144200 |
| 6000 | 1.567600 | 1.647648 |
| 7000 | 2.162200 | 2.242632 |
| 8000 | 2.822800 | 2.929152 |
| 9000 | 3.471200 | 3.707208 |
| 10000 | 4.372400 | 4.576800 |



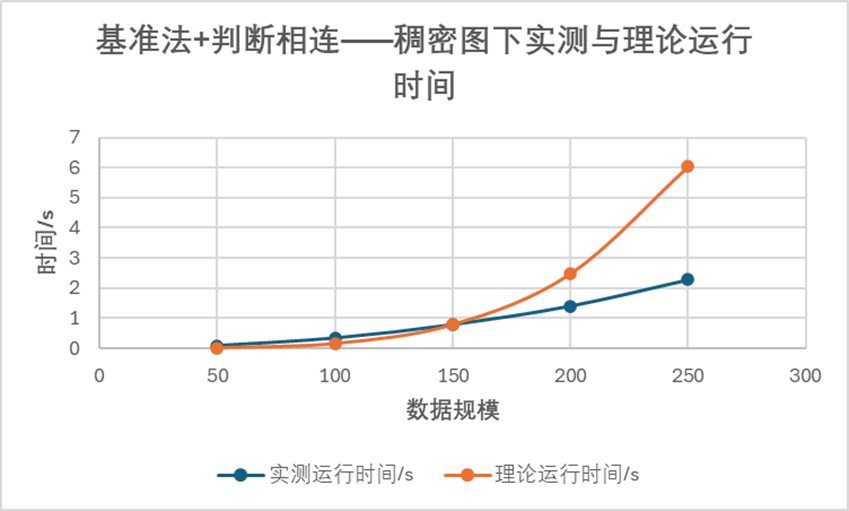
可以看到基本拟合，说明在稀疏图中基准算法+判断相连符合的时间复杂度。

对于稠密图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

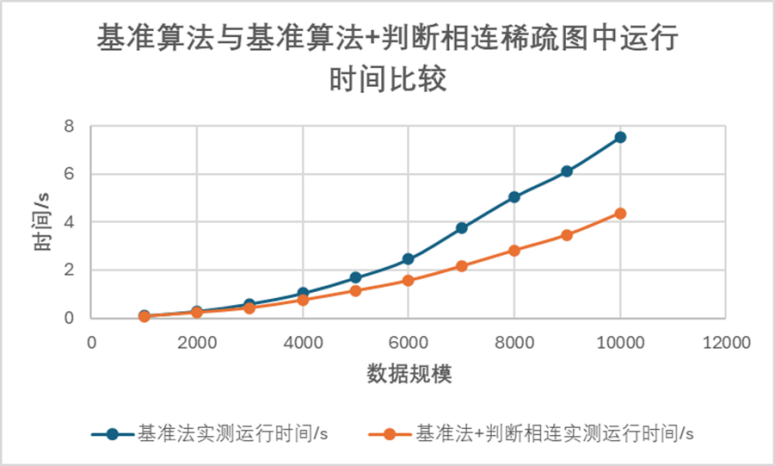
得到的数据如下

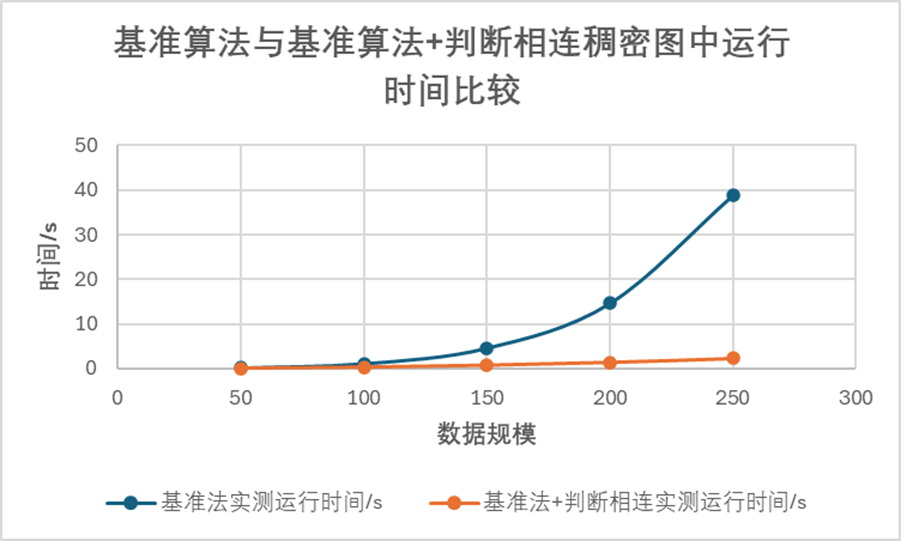
|  |  |  |
| --- | --- | --- |
| 基准法+判断相连——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 50 | 0.079200 | 0.009644 |
| 100 | 0.334600 | 0.154311 |
| 150 | 0.781200 | 0.781200 |
| 200 | 1.402600 | 2.468978 |
| 250 | 2.278600 | 6.027778 |



可以看到拟合效果并不好，显然实测运行时间低于理论运行时间，说明在稠密图中基准算法+判断相连的时间复杂度低于。

将基准算法与基准算法+判断相连实测运行时间进行比较，结果如下。





不难发现，判断相连的优化对基准法效率的提升很大，尤其是在稠密图中，优化后的算法相较于无优化的算法的运行速度有着质的提升。

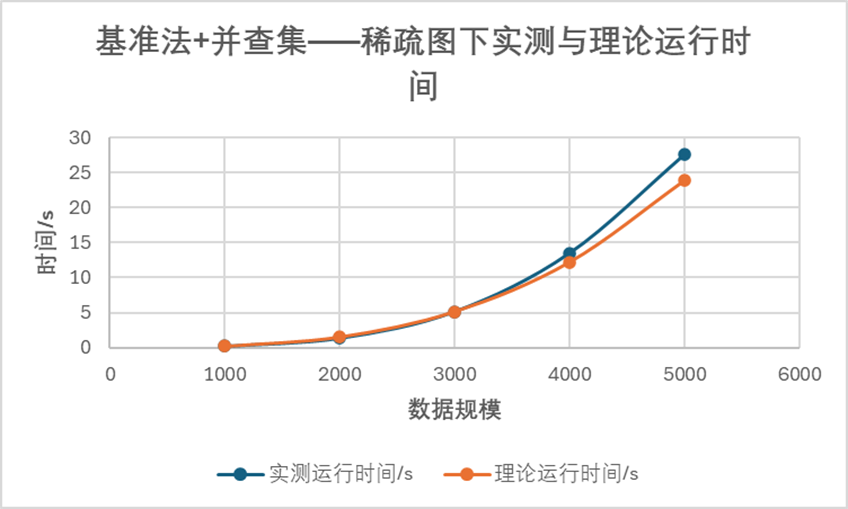
**（三）基准算法+并查集**

对于稀疏图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法+并查集——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.186000 | 0.190889 |
| 2000 | 1.359400 | 1.527111 |
| 3000 | 5.154000 | 5.154000 |
| 4000 | 13.488800 | 12.216889 |
| 5000 | 27.553000 | 23.861111 |



可以看到基本拟合，说明在稀疏图中基准算法+并查集符合的时间复杂度。

对于稠密图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法+并查集——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 50 | 0.172000 | 0.148152 |
| 100 | 4.626600 | 4.740872 |
| 150 | 36.001000 | 36.001000 |
| 200 | 156.097000 | 151.707918 |
| 250 | 478.125947 | 462.975823 |



可以看到基本拟合，说明在稠密图中基准算法+并查集符合的时间复杂度。

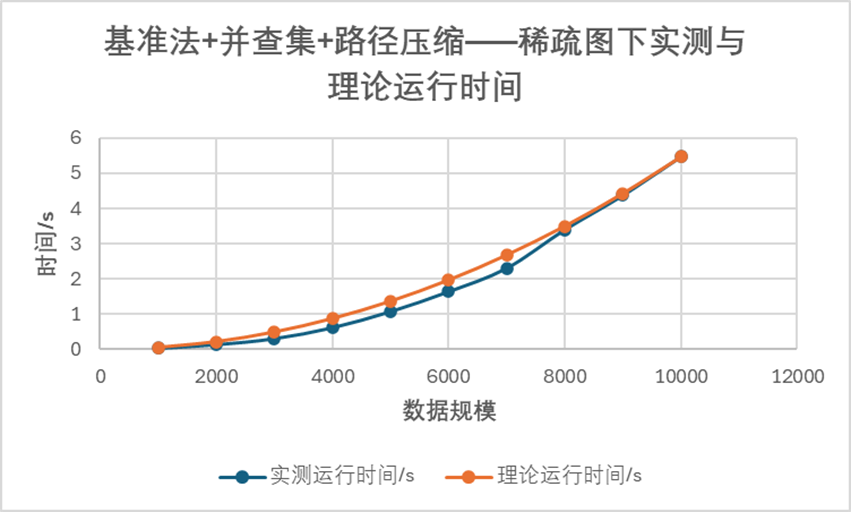
**（四）基准算法+并查集+路径压缩**

对于稀疏图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 基准法+并查集+路径压缩——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.036600 | 0.054642 |
| 2000 | 0.141000 | 0.218568 |
| 3000 | 0.311600 | 0.491778 |
| 4000 | 0.622200 | 0.874272 |
| 5000 | 1.076800 | 1.366050 |
| 6000 | 1.643600 | 1.967112 |
| 7000 | 2.306200 | 2.677458 |
| 8000 | 3.397400 | 3.497088 |
| 9000 | 4.371000 | 4.426002 |
| 10000 | 5.464200 | 5.464200 |



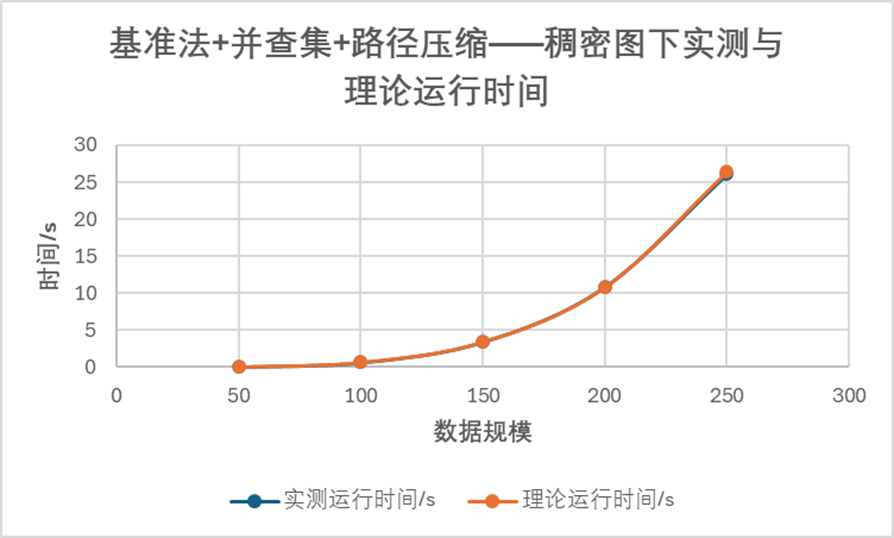
可以看到基本拟合，说明在稀疏图中基准算法+并查集+路径压缩符合的时间复杂度。

对于稠密图 ：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

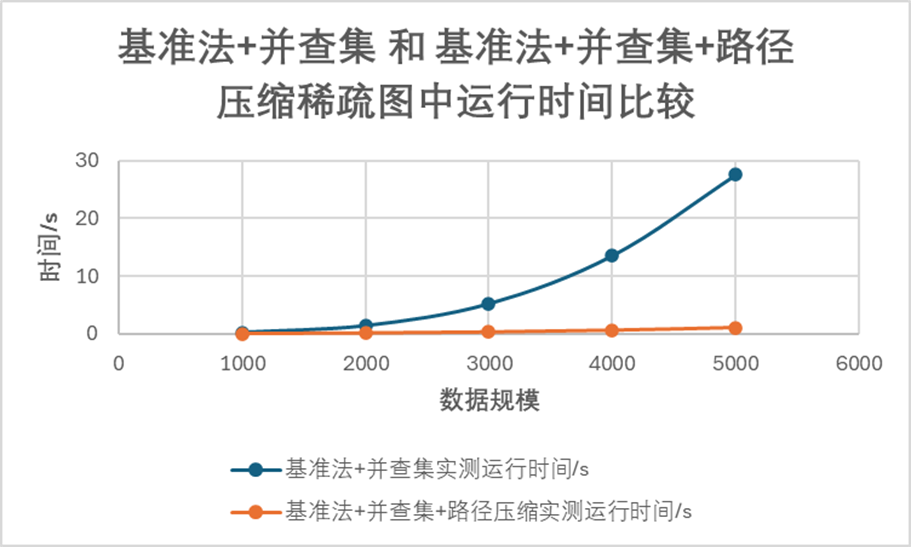
得到的数据如下

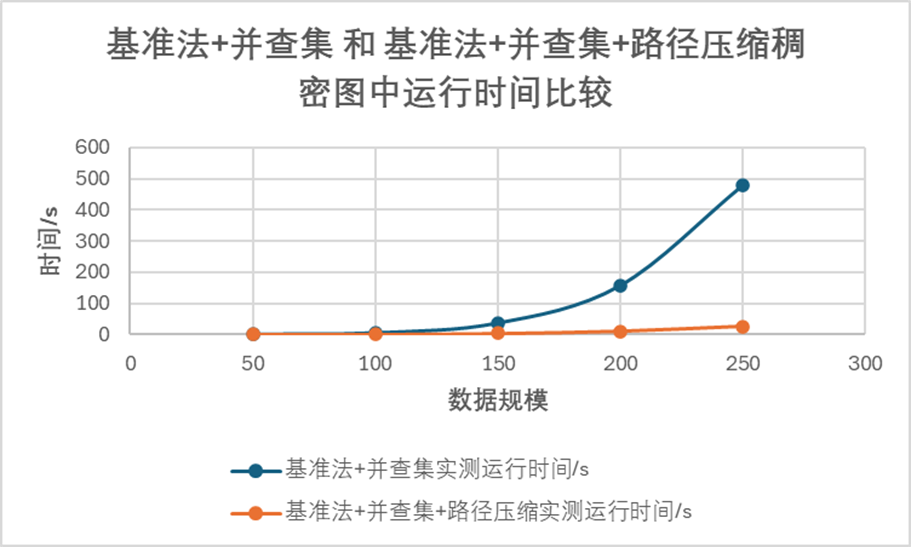
|  |  |  |
| --- | --- | --- |
| 基准法+并查集+路径压缩——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 50 | 0.040400 | 0.042207 |
| 100 | 0.641200 | 0.675319 |
| 150 | 3.418800 | 3.418800 |
| 200 | 10.853400 | 10.805096 |
| 250 | 26.112000 | 26.379630 |



可以看到基本拟合，说明在稠密图中基准算法+并查集+路径压缩符合的时间复杂度。

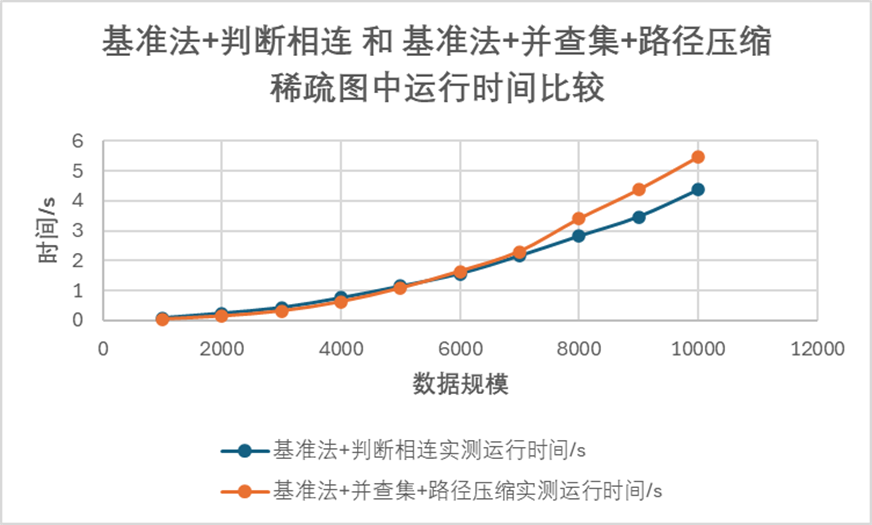
将基准算法+并查集与基准算法+并查集+路径压缩实测运行时间进行比较，结果如下。

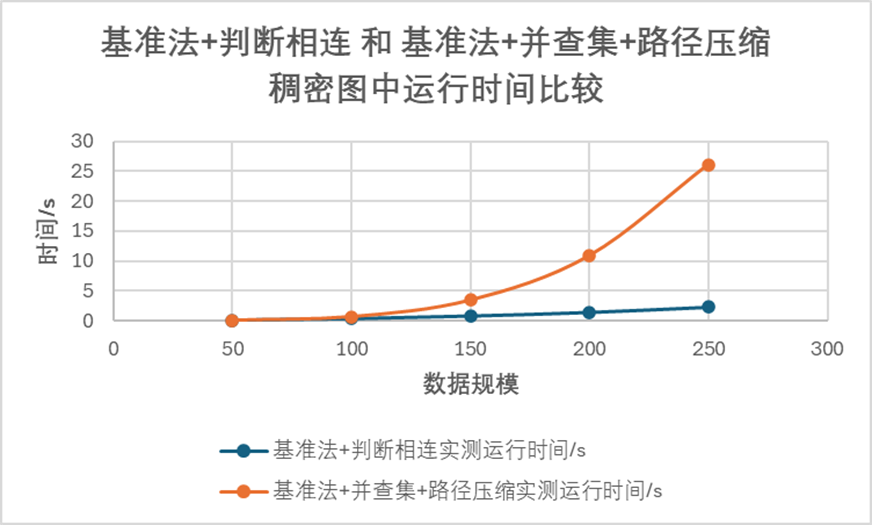




不难发现，路径压缩的优化对基准法+并查集算法在效率上提升很大。

将基准算法+判断相连与基准算法+并查集+路径压缩实测运行时间进行比较，结果如下。





不难发现，在稀疏图中，随着数据规模的增大，基准算法+并查集+路径压缩的运行速度逐渐快于基准算法+判断相连；在稠密图中，基准算法+判断相连的运行速度却远远快于基准算法+并查集+路径压缩，这是因为并查集在合并集合的过程中需要对图中每一条边进行遍历，图中边越多，会加大对基准算法+并查集+路径压缩的运行效率的下降，而基准算法+判断相连采用的是dfs，只要找到目标顶点即可回溯，不需要遍历图中所有边，故边越多，对基准算法+判断相连的效率影响不大，因此在稠密图中会出现基准算法+判断相连运行效率高于基准算法+并查集+路径压缩运行效率的情况。

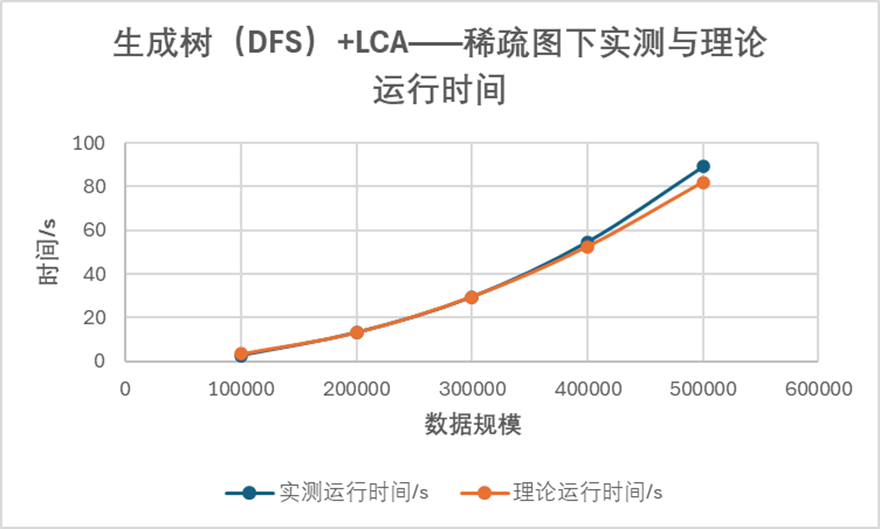
**（五）生成树+LCA**

对于稀疏图 、用DFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（DFS）+LCA——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 100000 | 2.663600 | 3.278844 |
| 200000 | 13.158600 | 13.115378 |
| 300000 | 29.509600 | 29.509600 |
| 400000 | 54.709400 | 52.461511 |
| 500000 | 89.235400 | 81.971111 |



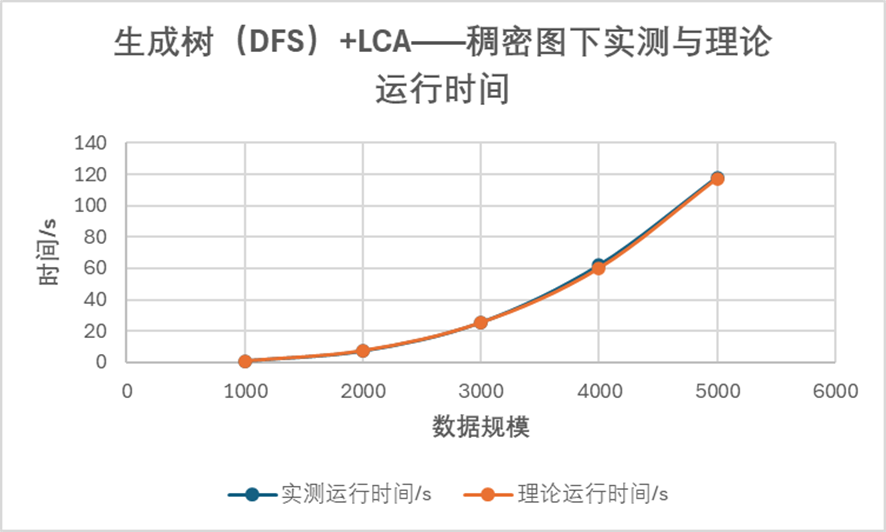
可以看到基本拟合，说明在稀疏图中生成树（DFS）+LCA符合的时间复杂度。

对于稠密图 、用DFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（DFS）+LCA——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.854000 | 0.938919 |
| 2000 | 7.187800 | 7.511348 |
| 3000 | 25.350800 | 25.350800 |
| 4000 | 61.930000 | 60.090785 |
| 5000 | 118.112000 | 117.364815 |



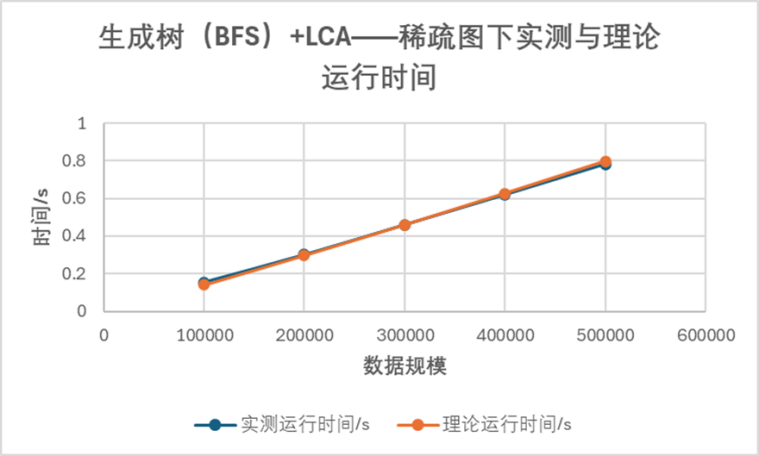
可以看到基本拟合，说明在稠密图中生成树（DFS）+LCA符合的时间复杂度。

对于稀疏图 、用BFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（BFS）+LCA——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 100000 | 0.155200 | 0.139915 |
| 200000 | 0.301200 | 0.296678 |
| 300000 | 0.459800 | 0.459800 |
| 400000 | 0.619600 | 0.627051 |
| 500000 | 0.781800 | 0.797373 |



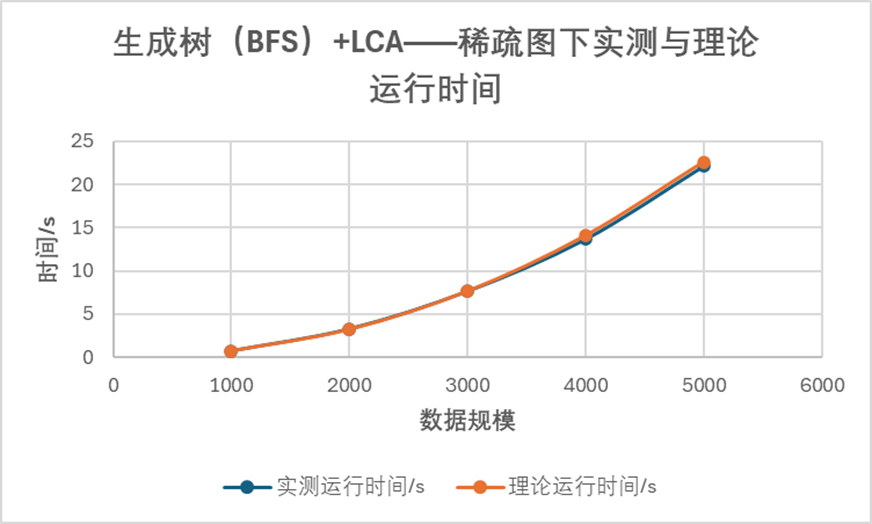
可以看到基本拟合，说明在稀疏图中生成树（BFS）+LCA符合的时间复杂度。

对于稠密图 、用BFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

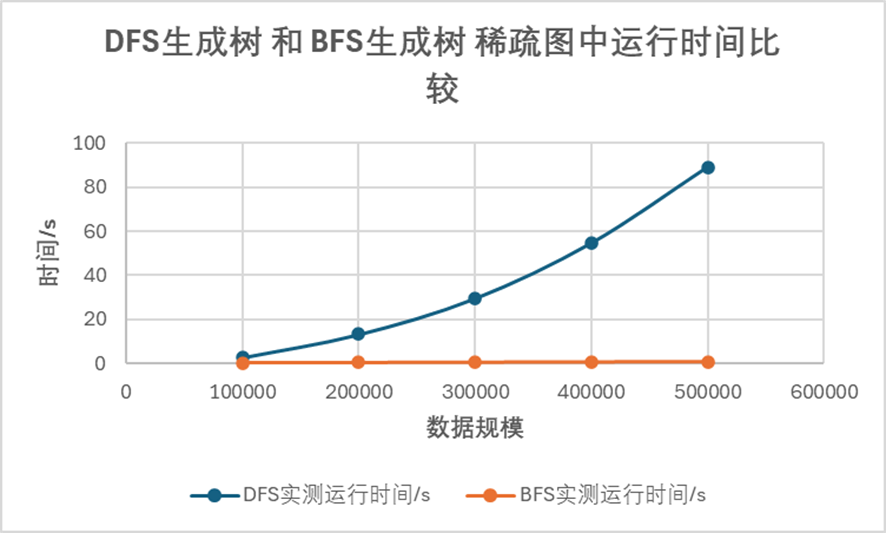
得到的数据如下

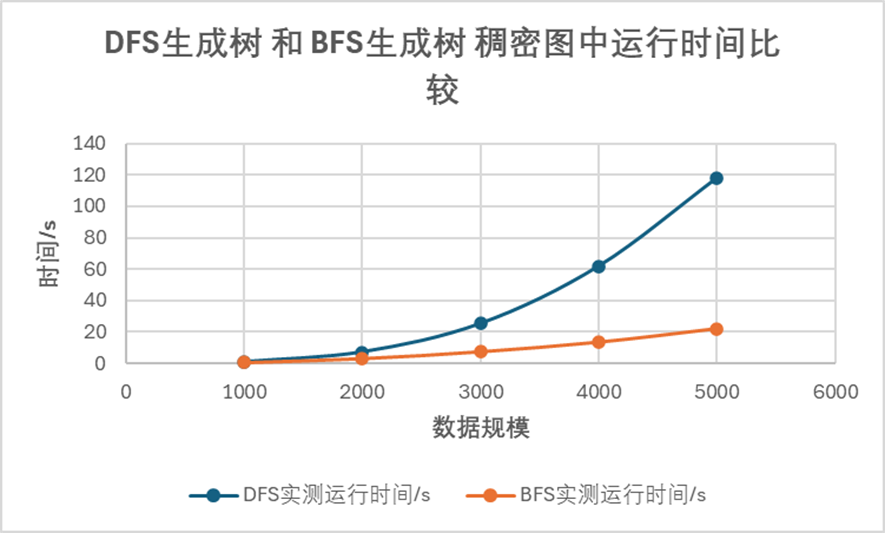
|  |  |  |
| --- | --- | --- |
| 生成树（BFS）+LCA——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.724200 | 0.731793 |
| 2000 | 3.272800 | 3.220895 |
| 3000 | 7.633600 | 7.633600 |
| 4000 | 13.654000 | 14.058467 |
| 5000 | 22.112000 | 22.557339 |



可以看到基本拟合，说明在稠密图中生成树（BFS）+LCA符合的时间复杂度。

对比生成树+LCA在DFS和BFS两个不同生成树方法下实测运行时间。结果如下。





不难发现，用BFS生成树的效率通常高于用DFS生成树的效率，可能是因为BFS生成的树深度较浅，各分支长度相对均衡，这有利于加快后续向上查找最近公共祖先（LCA）的过程。而DFS生成的树深度较深，在查找LCA时需要更多的时间向上探查，因此相对效率较低。

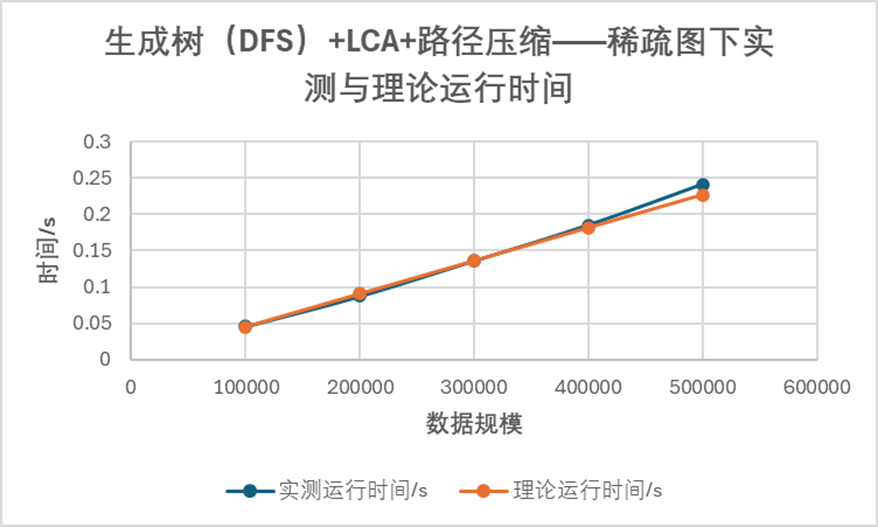
**（六）生成树+LCA+路径压缩**

对于稀疏图 、用DFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（DFS）+LCA+路径压缩——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 100000 | 0.046000 | 0.045333 |
| 200000 | 0.087400 | 0.090667 |
| 300000 | 0.136000 | 0.136000 |
| 400000 | 0.185200 | 0.181333 |
| 500000 | 0.241200 | 0.226667 |



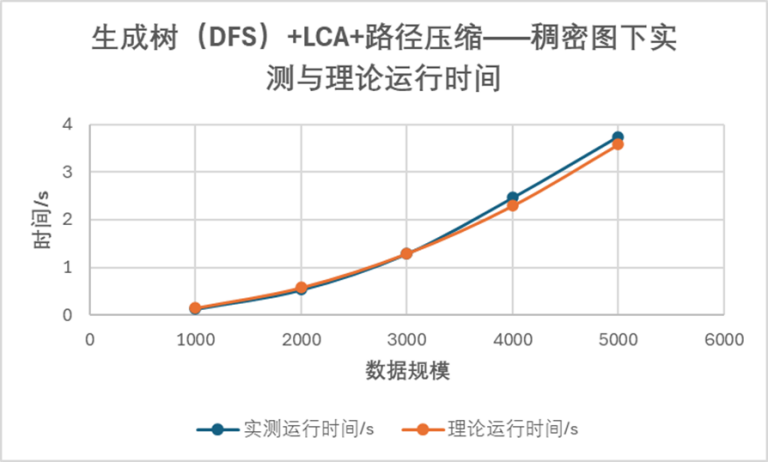
可以看到基本拟合，说明在稀疏图中生成树（DFS）+LCA+路径压缩符合的时间复杂度。

对于稠密图 、用DFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（DFS）+LCA+路径压缩——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.130400 | 0.143222 |
| 2000 | 0.533000 | 0.572889 |
| 3000 | 1.289000 | 1.289000 |
| 4000 | 2.462200 | 2.291556 |
| 5000 | 3.745200 | 3.580556 |



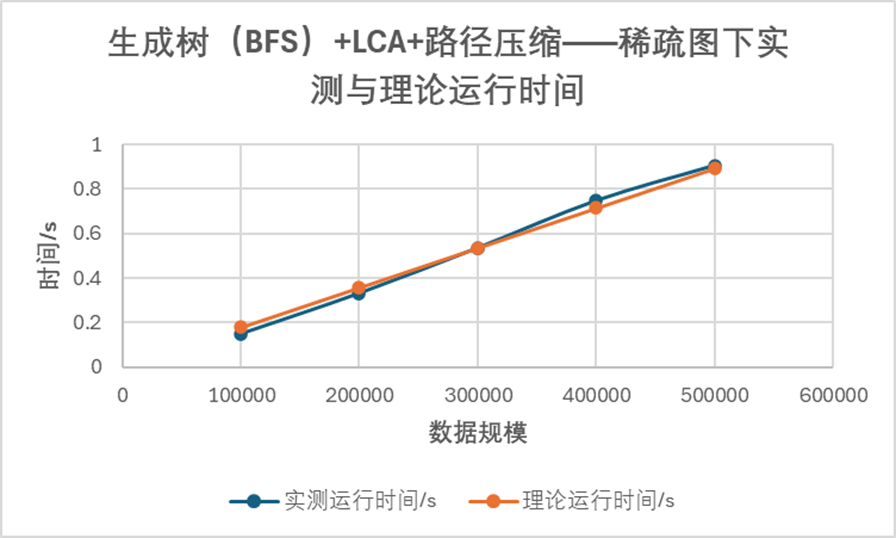
可以看到基本拟合，说明在稠密图中生成树（DFS）+LCA+路径压缩符合的时间复杂度。

对于稀疏图 、用BFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

得到的数据如下

|  |  |  |
| --- | --- | --- |
| 生成树（BFS）+LCA+路径压缩——稀疏图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 100000 | 0.150400 | 0.178600 |
| 200000 | 0.332600 | 0.357200 |
| 300000 | 0.535800 | 0.535800 |
| 400000 | 0.748000 | 0.714400 |
| 500000 | 0.905800 | 0.893000 |



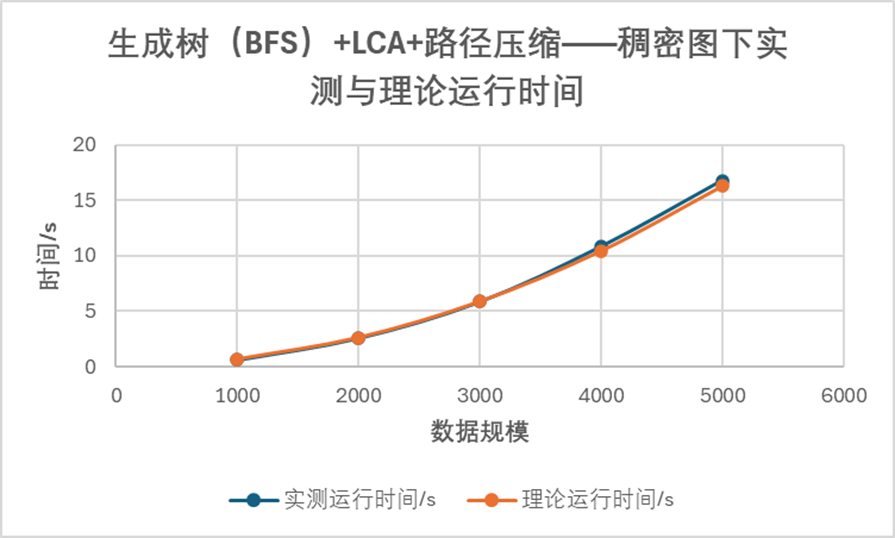
可以看到基本拟合，说明在稀疏图中生成树（BFS）+LCA+路径压缩符合的时间复杂度。

对于稠密图 、用BFS建立生成树：

生成多组随机数据，每组数据运行 10 次并取平均值，理论运行时间的计算方程为

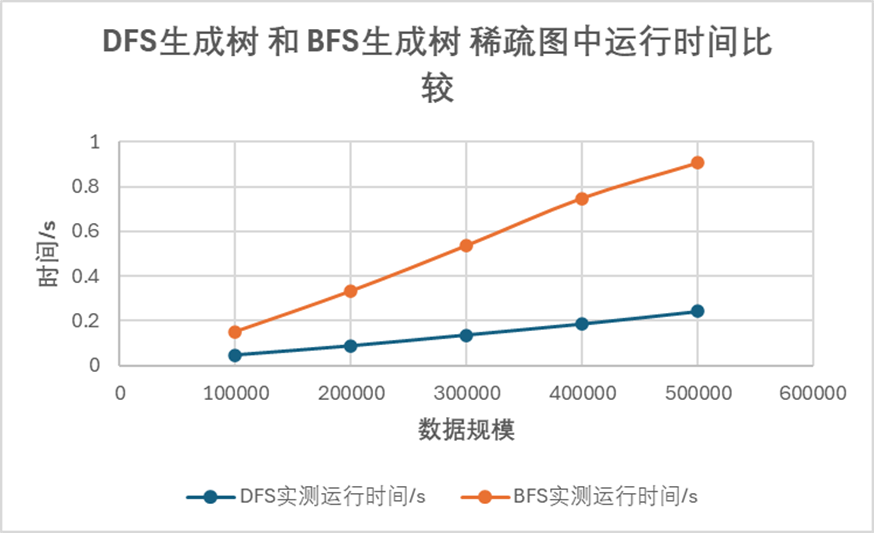
得到的数据如下

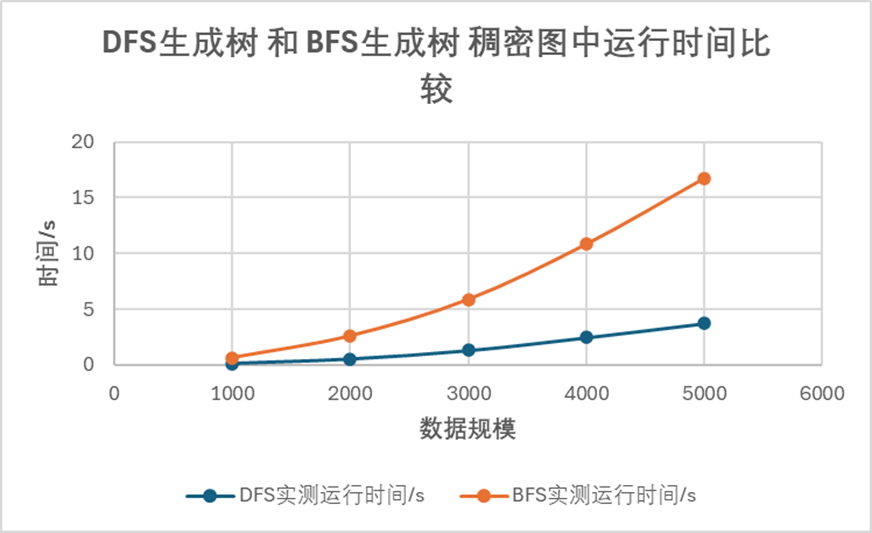
|  |  |  |
| --- | --- | --- |
| 生成树（BFS）+LCA+路径压缩——稠密图下实测与理论运行时间 | | |
| 顶点数据规模n | 实测运行时间/s | 理论运行时间/s |
| 1000 | 0.639200 | 0.651756 |
| 2000 | 2.597600 | 2.607022 |
| 3000 | 5.865800 | 5.865800 |
| 4000 | 10.829000 | 10.428089 |
| 5000 | 16.754200 | 16.293889 |



可以看到基本拟合，说明在稠密图中生成树（BFS）+LCA+路径压缩符合的时间复杂度。

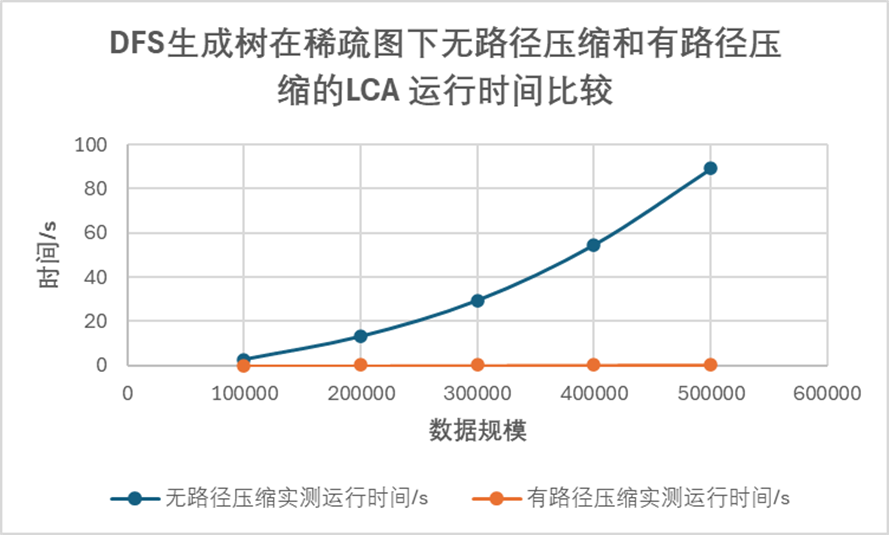
对比生成树+LCA+路径压缩在DFS和BFS两个不同生成树方法下实测运行时间。结果如下。

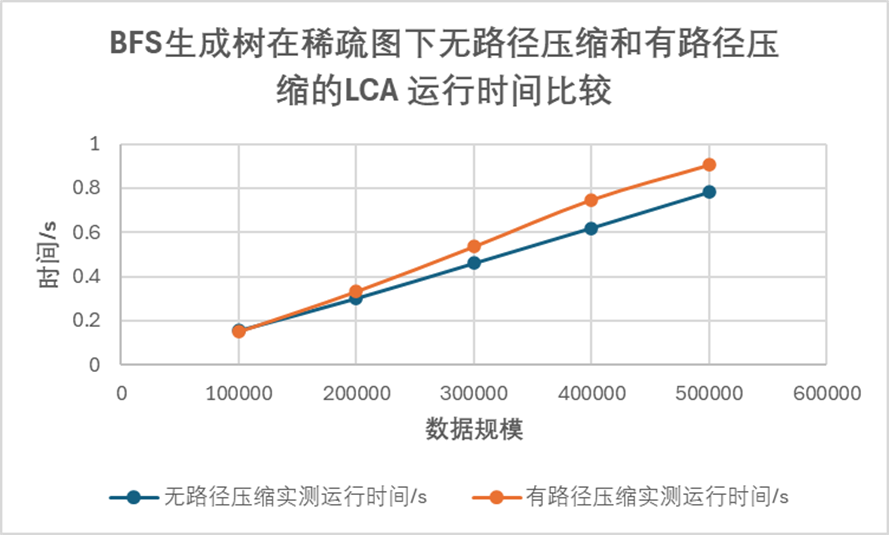


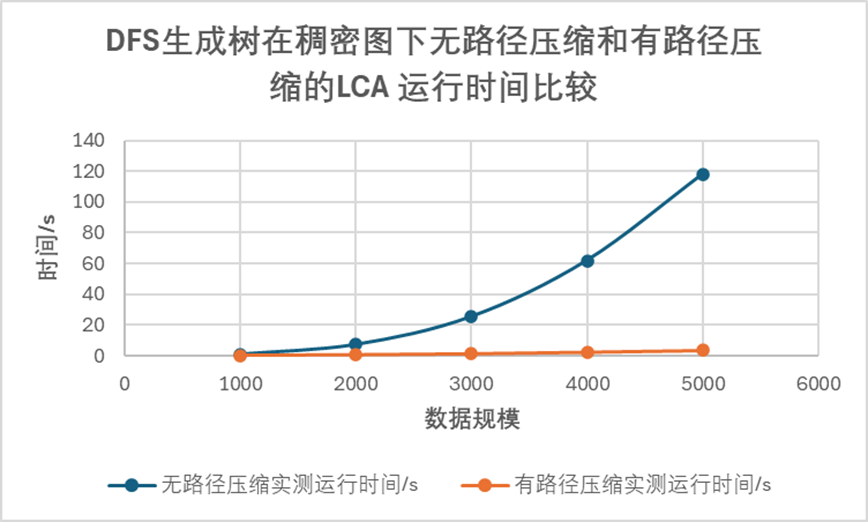


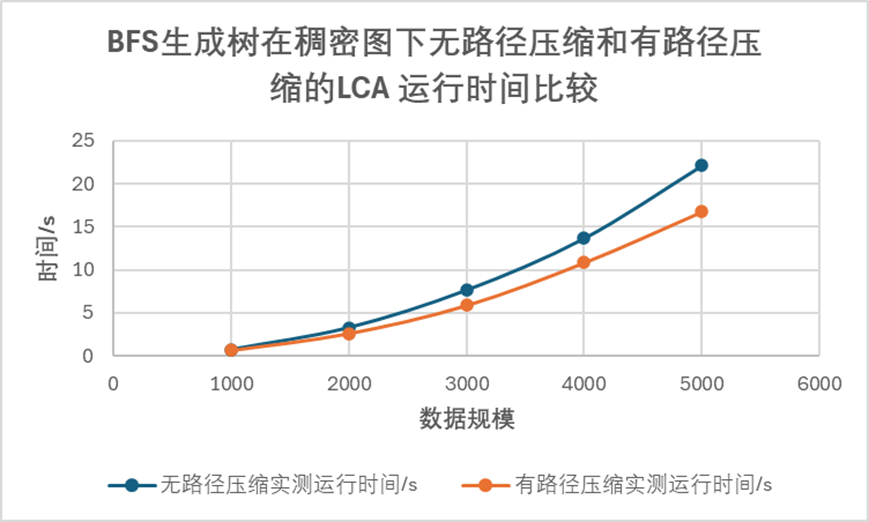
不难发现，在生成树+LCA+路径压缩算法下，用DFS生成树的运行效率通常高于用BFS生成树的运行效率，可能是因为DFS生成的树通常较深，路径压缩能够有效地扁平化这些较深的树，从而显著优化向上探查LCA操作的性能，而用BFS生成的树较浅，树的高度本身已经较低，路径压缩的优化效果不如在深度较大的DFS生成树中显著，因此在结合路径压缩时，DFS生成树的整体效率更高。

接下来对比生成树+LCA算法在无路径压缩和有路径压缩下的实测运行时间。





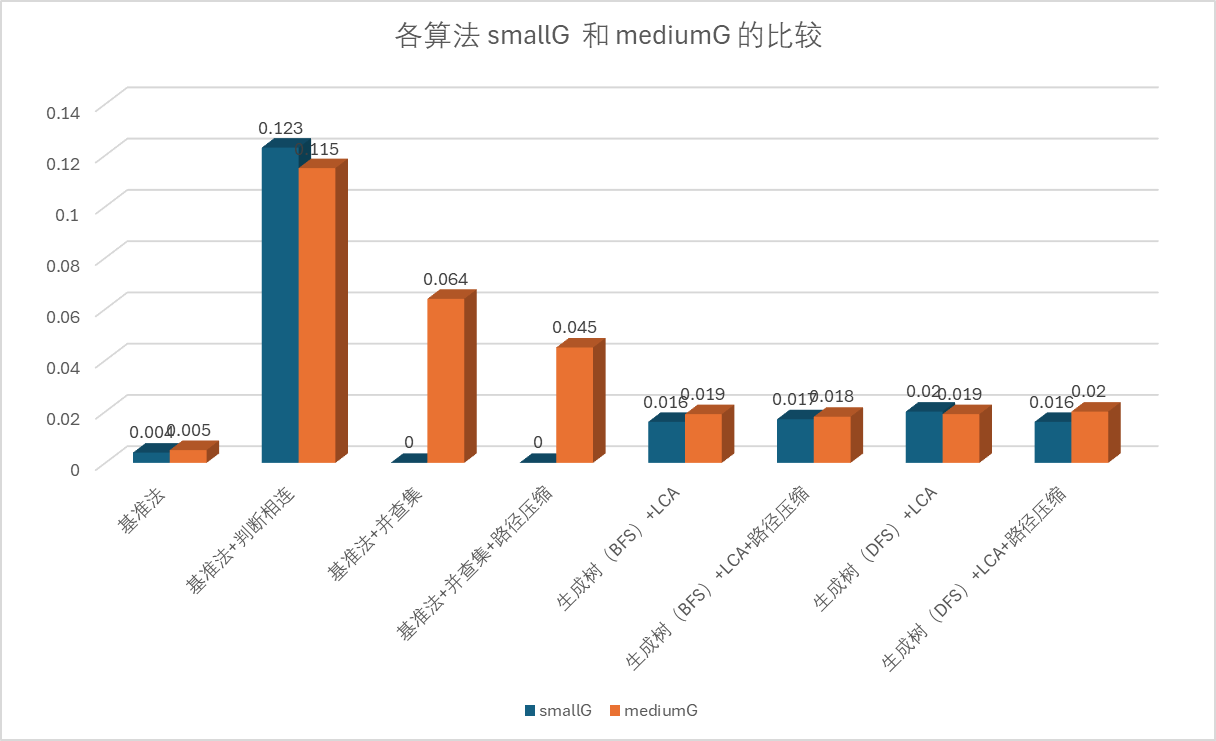


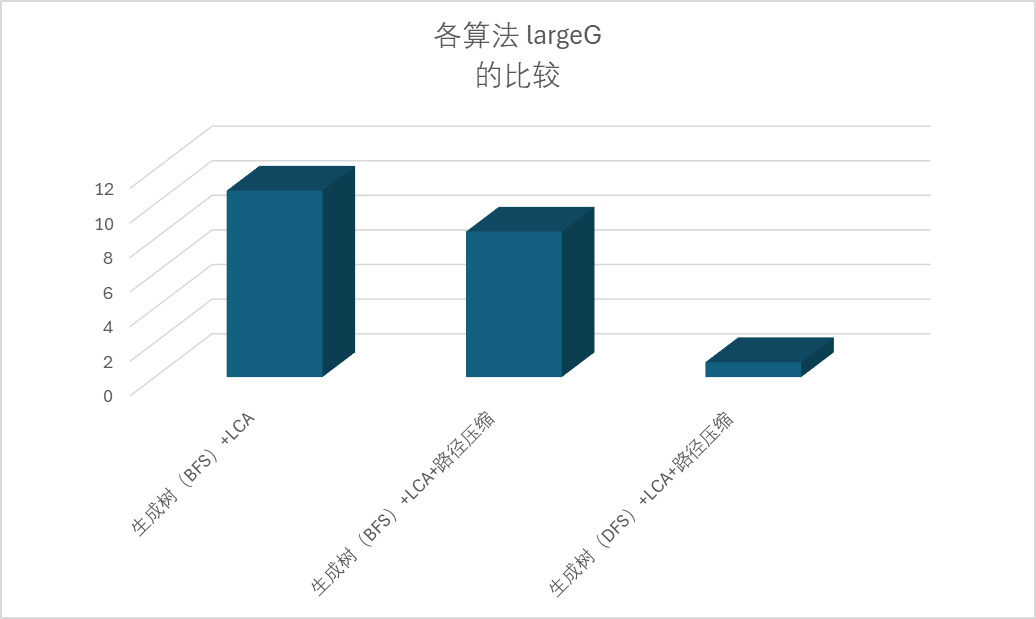


可以发现，无论在稀疏图还是稠密图中，路径压缩对采用DFS生成树方法的算法效率提升显著，因为DFS生成的树深度较深，路径压缩能够显著降低树的高度，从而提高算法效率。而对于采用BFS生成树方法的算法，路径压缩提升效果不大，甚至在稠密图中可能会降低效率，这是因为BFS生成的树较浅，树上的路径已经较短，路径压缩反而带来了额外的开销。因此，路径压缩对DFS生成树方法的效率提升优于对BFS生成树方法的提升。

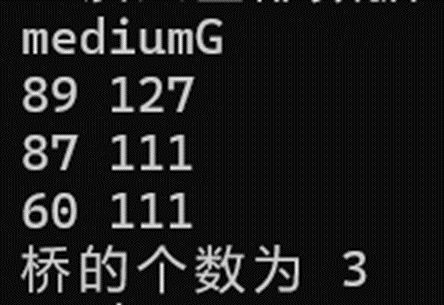
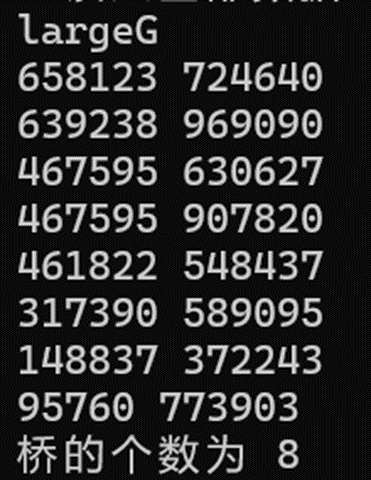
**（七）各算法在smallG、mediumG、largeG中的表现**

|  |  |  |  |
| --- | --- | --- | --- |
|  | smallG运行时间/s | mediumG运行时间/s | largeG运行时间/s |
| 基准法 | 0.004000 | 0.005000 |  |
| 基准法+判断相连 | 0.123000 | 0.115000 |  |
| 基准法+并查集 | 0 | 0.064000 |  |
| 基准法+并查集+路径压缩 | 0 | 0.045000 |  |
| 生成树（BFS）+LCA | 0.016000 | 0.019000 | 10.776900 |
| 生成树（BFS）+LCA+路径压缩 | 0.017000 | 0.018000 | 8.409000 |
| 生成树（DFS）+LCA | 0.020000 | 0.019000 |  |
| 生成树（DFS）+LCA+路径压缩 | 0.016000 | 0.020000 | 0.862000 |





smallG、mediumG、largeG得到的桥边和桥数如下。

**七、经验总结**

这次实验最大的收获是，在求解问题时可以采用排除法来优化算法，从直接解决问题转变为探究问题的反面。例如，在实验中通过识别不是桥的边（即环边），间接地找到所有桥边，这种方法使得可以采用更高效的算法来解决问题，从而提升了整体的解决效率。同时，还学会了用VS中的O2优化来加快程序运行，调整栈空间大小来保证DFS建树不会造成栈溢出。

|  |
| --- |
| 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。