最小生成树算法的对比研究

范皓年 1900012739
袁世平 1700012899
北京大学 信息科学技术学院

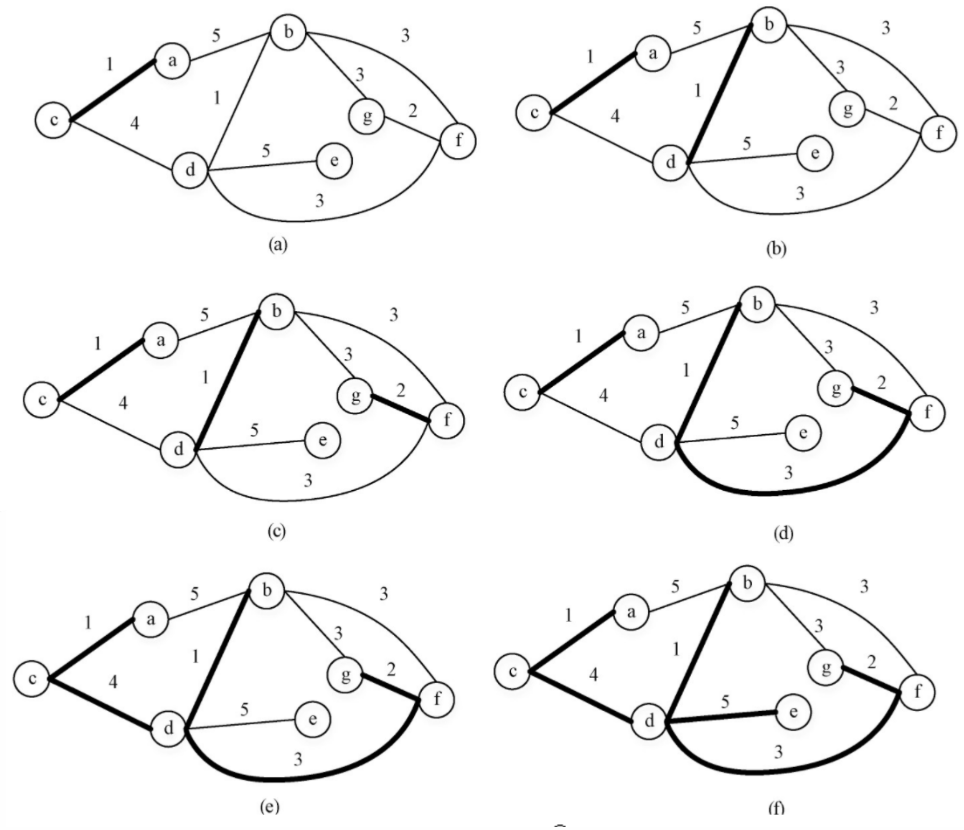
最小生成树：背景

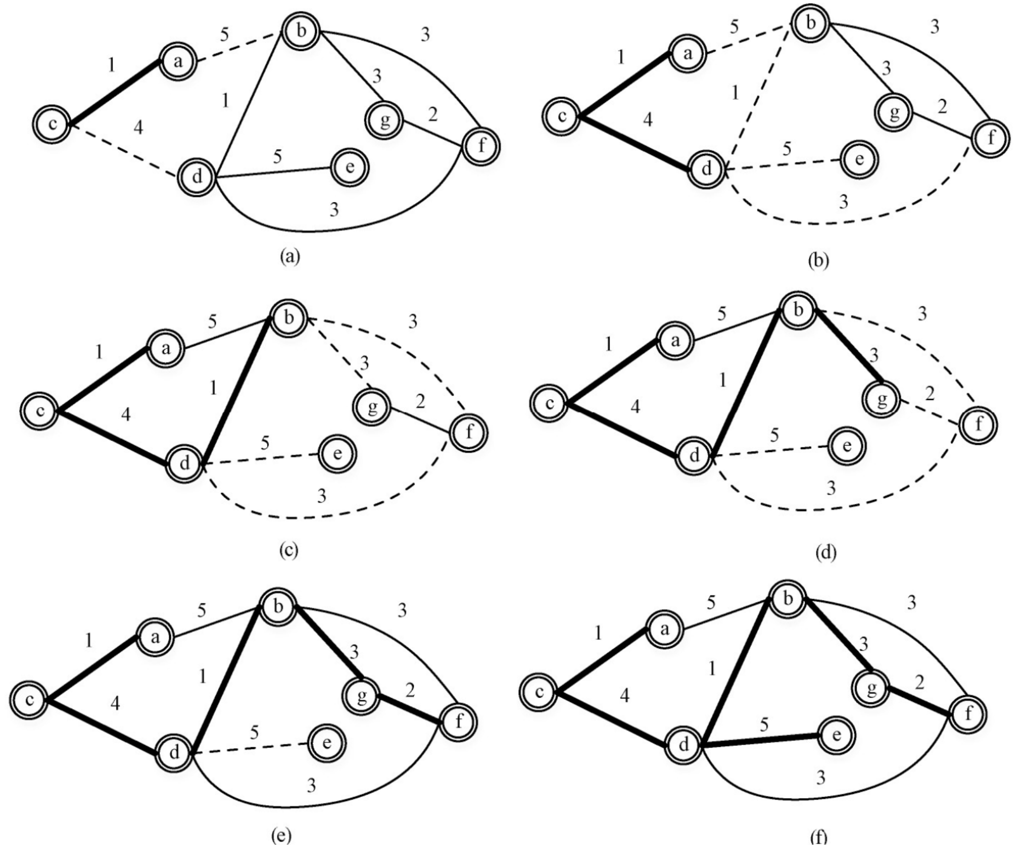
- 网络无处不在，对于已有的稍见规模的社会系统，和现在逐渐发展蓬勃的互联网系统，都存在连接和组织的问题。为了联系一个网络系统当中的各个节点，我们需要构建**连通图**。
- 在实际生活中，由于工程难度和用户协调等等问题，成本相差极大，构建不同方案的连通图将会有较大的财力和时间成本的差异，所以这个问题有着重大的研究价值。

最小生成树：引入

- 对于n个节点组成的图，所有可能的连接共$\frac{n(n-2)}{2}$条，为构建一个边权最低的连通图，我们需要从中选取$\mathbf{n-1}$条权值最低的边，求解一个连通图的最小连通分支的问题就是**最小生成树**问题。
- 已有的比较成熟的最小生成树算法有“Prim算法”，“Kruskal算法”，以及时常为了减小Prim的遍历的时间开销而进行优化的“带堆优化的Prim算法”。由于Prim算法在绝大多数稀疏度下都能表现出明显优势，所以我们这里不再研究堆优化Prim。利用Prim和Kruskal两种算法，对图中各边进行等权、排列权、随机权值的实验验证，探讨常见的MST算法在应用中的优劣。

算法描述：算法的实现过程





[1]贺军忠,王丽君.Kruskal和Prim算法的分析研究与比较[J].陇东学院学报,2020,31(02):8-11.

算法证明：Kruskal和Prim

伪代码：

Input. The edges of the graph e, where each element in e is (u, v, w) denoting that there is an edge between u and v weighted w.
Output. The edges of the MST of the input graph.
Method.
$result \leftarrow \varnothing$
sort e into nondecreasing order by weight w
for each (u, v, w) in the sorted e
    if u and v are not connected in the union-find set
        connect u and v in the union-find set
        $result \leftarrow result \bigcup \{(u, v, w)\}$
return $result$

Input. The nodes of the graph V ; the function $g(u, v)$ which means the weight of the edge (u, v); the function $adj(v)$ which means the nodes adjacent to v.
Output. The sum of weights of the MST of the input graph.
Method.
$result \leftarrow 0$
choose an arbitrary node in V to be the $root$
$dis(root) \leftarrow 0$
for each node $v \in (V - \{root\})$
    $dis(v) \leftarrow \infty$
$rest \leftarrow V$
while $rest \neq \varnothing$
    $cur \leftarrow$ the node with the minimum dis in $rest$
    $result \leftarrow result + dis(cur)$
    $rest \leftarrow rest - \{cur\}$
    for each node $v \in adj(cur)$
        $dis(v) \leftarrow \min(dis(v), g(cur, v))$
return $result$

Kruskal: O(|E|·lg|E|)

Prim: O(|V|·|E|)

实验设计

- 枚举节点 N [100 - 1000] [2000 - 20000]

- 枚举边的数量 M
  - 极端情况：树 [N-1]、完全图 [N * (N-1) / 2]
  - 如何生成中间的稠密度？
    - ✓1. 分散加边
    - ✓2. 集中加边

- 探究边权的影响
  - 权值中相等的值可能对排序和选最小值有影响
    - ✓等权、排列、随机

实验实现 - 环境介绍

- 实验环境： Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-142-generic x86_64) 48核
- 测速方法：

```
#include<sys/time.h>
#include<time.h>
double get_time(){
    struct timeval tv;
    double t;
    gettimeofday(&tv, (struct timezone *)0);
    t = tv.tv_sec + (double)tv.tv_usec * 1e-6;
    return t;
}
```

- <sys/time.h> gettimeofday() 函数， 精确到微秒 (10^{-6} s)
- 忽略读入时间， 测试10次取平均值

实验实现-生成边权

- 分三种方式生成边权

  - 等权图，所有边的权值相同
    - ✓ 直接令所有边权为1
  - 排列图，所有边的权值不相同
    - ✓ 构造数组，然后调用<algorithm>中的random_shuffle函数
  - 随机图，所有边的权值随机
    - ✓ 使用<cstdlib>库中的伪随机数获取(1,m)之间的随机数

实验实现-构图

建树

```
// 生成树
// 枚举左节点
for u from 2 to n
    // 枚举一个[1, u-1]内的随机数作为右节点
    v = rand() % (u - 1) + 1
    add_edge(u, v)
```

- 随机一个比自己小的节点，可以保证自己连通的同时不会出现环

构建完全图

```
// 生成完全图
// 枚举左节点
for u from 1 to n
    // 枚举右节点
    for v from u + 1 to n
        add_edge(u, v)
```

实验实现-构图

- 构建中间稠密度的图，分轮次T给初始为树的图加边
  - 平均加边：每次给度数 <T 的节点加边

```
// 平均方式加边
function: add_average(T)
    //枚举左节点
    for u from 1 to n
        // 选择度数小于T的节点加边
        if deg[u] < T
            for v from 1 to n
                // 找到一个度数小于T，不曾连过边的不同节点连接
                if deg[v] < T and !connect(u,v) and u != v
                    add_edge(u, v)
                    // 一个节点只主动加一条
                    break
```

  - 集中加边：每次给一个节点加满边

```
// 集中方式加边
function: add_pernode(T)
    // 第T次就把第T个节点补满
    u = T
    for v from 1 to n
        // 找到所有u没有连接到的节点都连接
        if u != v and !connect(u,v)
            add_edge(u, v)
```

实验实现-测试

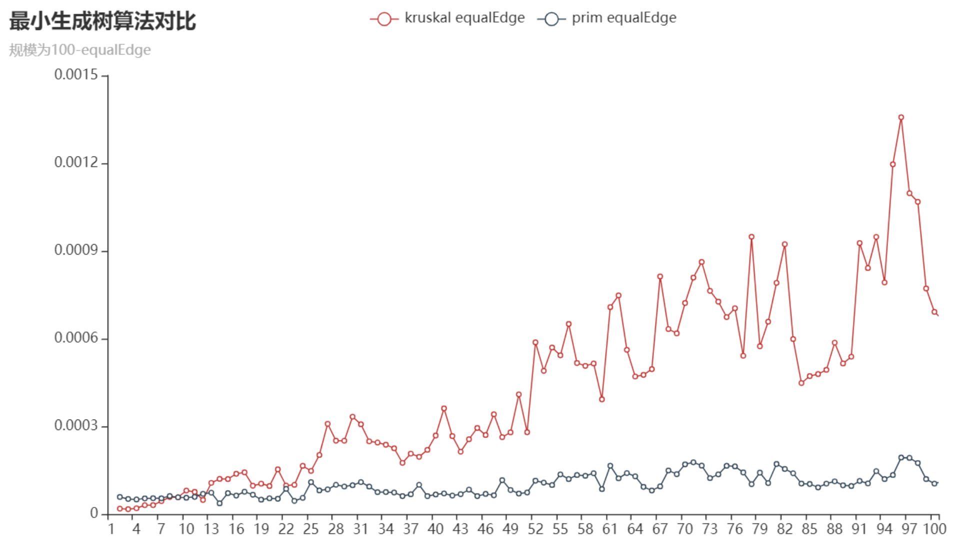
- 构造脚本进行自动测试

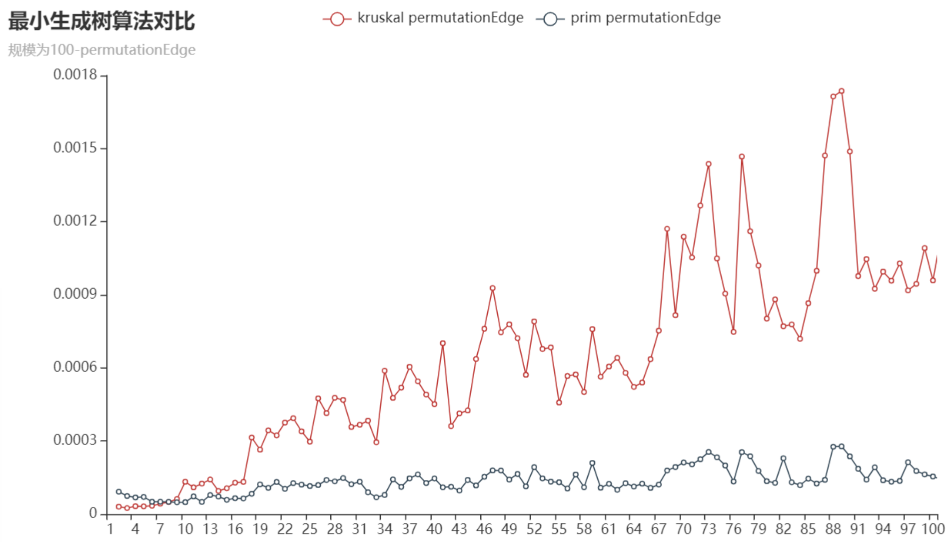
```
枚举节点N
    根据N得到M
    并根据M生成边权
    枚举不同的边权
        根据当前边权生成图
        枚举不同的图
            枚举进行的轮数
                根据当前的图执行prim算法
            计算平均值
            枚举进行的轮数
                根据当前的图执行kruskal算法
            计算平均值
```

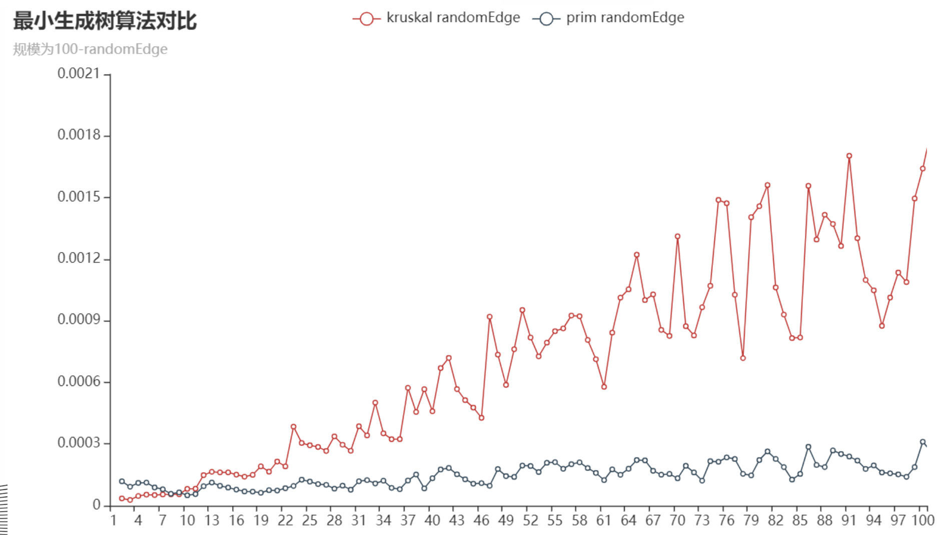
```
g++ -g createGraph.cpp -o createGraph.o
g++ -g prim.cpp -o prim.o
g++ -g kruskal.cpp -o kruskal.o
g++ -g getaverage.cpp -o getaverage.o

mkdir -p testData
mkdir -p recordData
for n in 2000 5000 10000
do
    tmp=`expr $n - 1`
    echo $tmp
    sum=`expr $n \* $tmp`
    sum=`expr $sum / 2`
    mod=`expr $n / 50`
    echo $mod
    sh -c './createEdge.o '$sum''
    for filename in equalEdge.txt permutationEdge.txt randomEdge.txt
    do
        sh -c './createGraph.o '$filename' '$n' '$sum''
        for graph in `seq 0 2 200`
        do
            echo "" >  ./testData/tmp.txt
            echo $graph 'prim'
            for T in `seq 0 9`
            do
                echo 'test' $T
                sh -c './prim.o < ./'$n'pernode/graph'$graph'.txt >> ./testData/tmp.txt'
            done
            sh -c './getaverage.o > ./recordData/prim_'$n'_'$filename'_'$graph'.txt'
            echo $graph kruskal
            echo "" > ./testData/tmp.txt
            for T in `seq 0 9`
            do
                echo 'test' $T
                sh -c './kruskal.o < ./'$n'pernode/graph'$graph'.txt >> ./testData/tmp.txt'
            done
            sh -c './getaverage.o > ./recordData/kruskal_'$n'_'$filename'_'$graph'.txt'
        done
    done
    rm -r $n*
done
```

实验过程：稀疏度验证

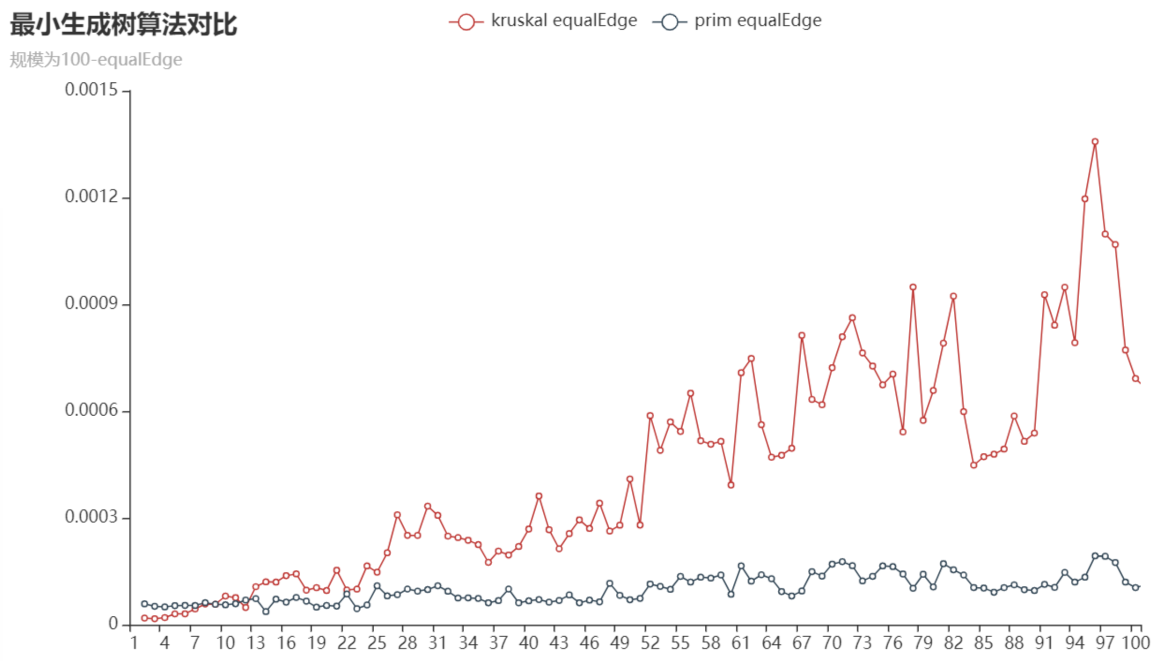

最小生成树算法对比
规模为100-equalEdge
kruskal equalEdge
prim equalEdge


最小生成树算法对比
规模为100-permutationEdge
kruskal permutationEdge
prim permutationEdge


最小生成树算法对比
规模为100-randomEdge
kruskal randomEdge
prim randomEdge

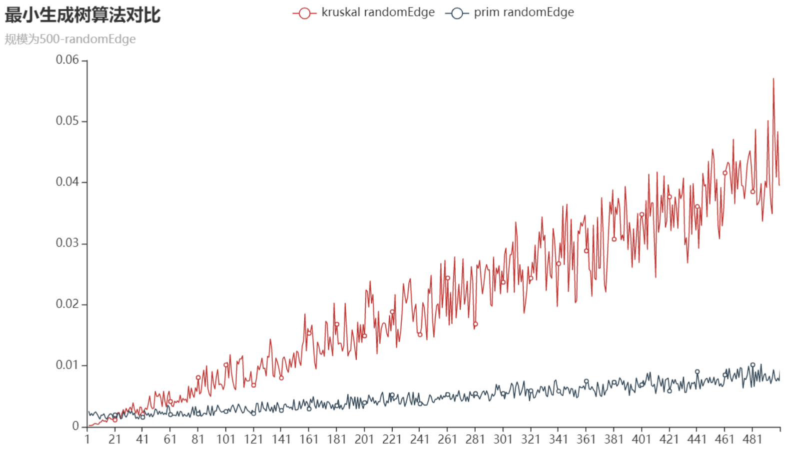
实验过程：稀疏度验证

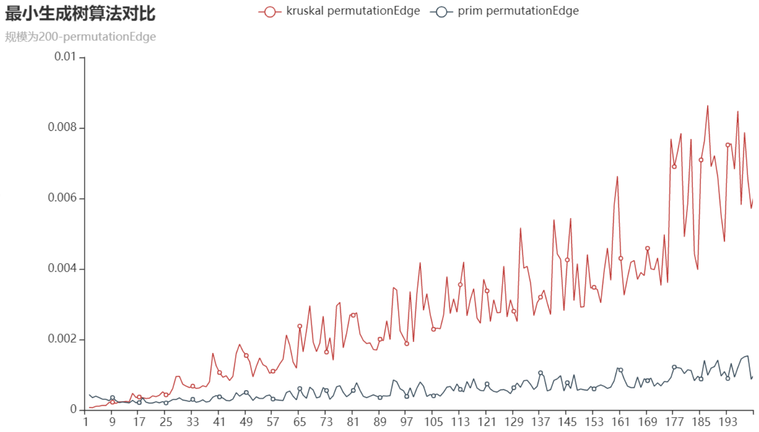
- 大体确定复杂度之后，为了显示出tradeoff的全过程，我们先进行粗筛，从而决定研究范围。
- 如右图，可以看出prim几乎完胜。但在叫稀疏的图中kruskal由于边数较少，所以表现出一定的优势。
- 进一步验证：

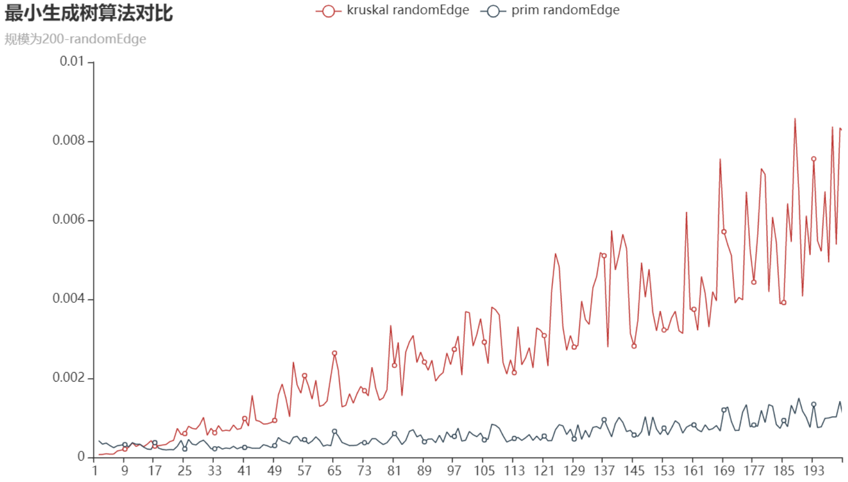


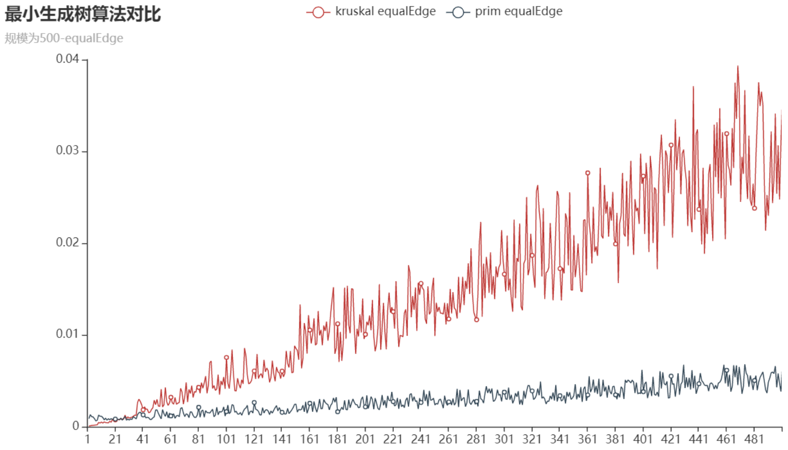
实验过程：稀疏度验证

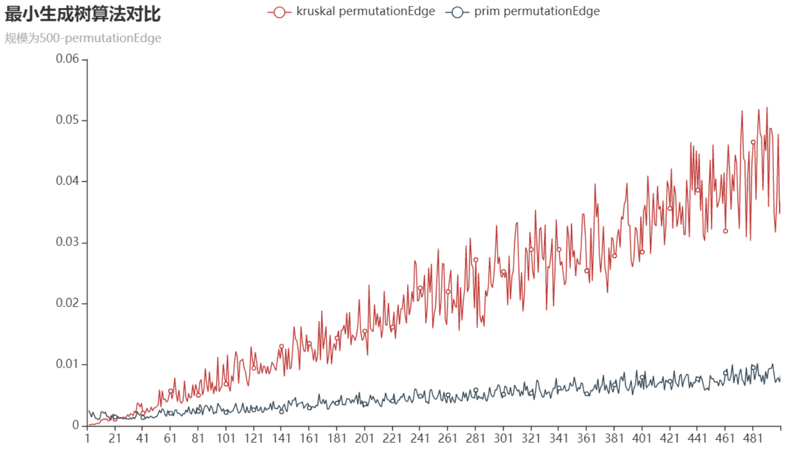
■进一步验证

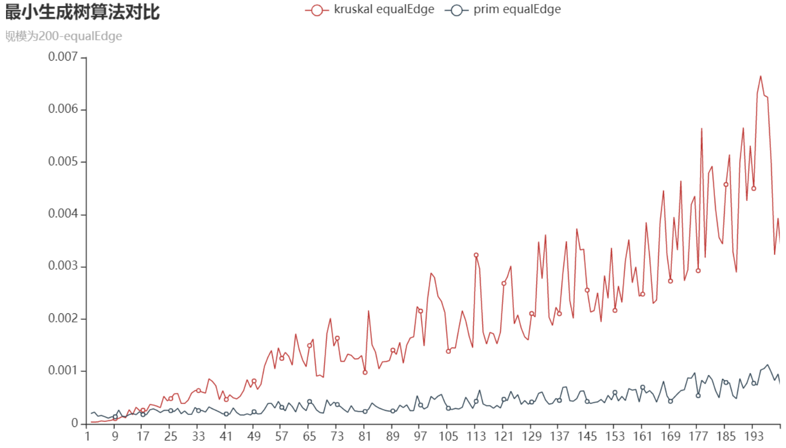






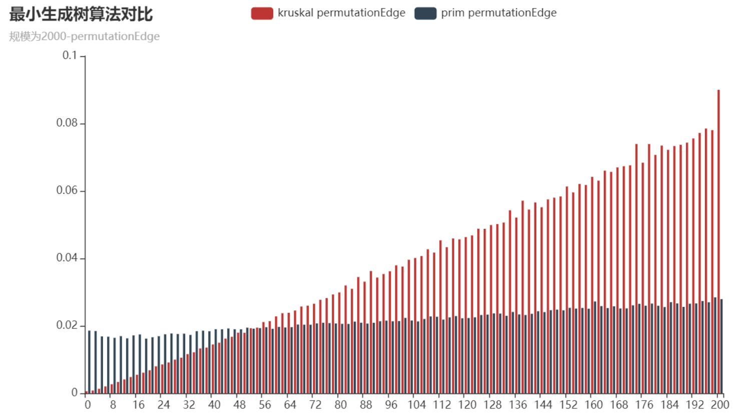


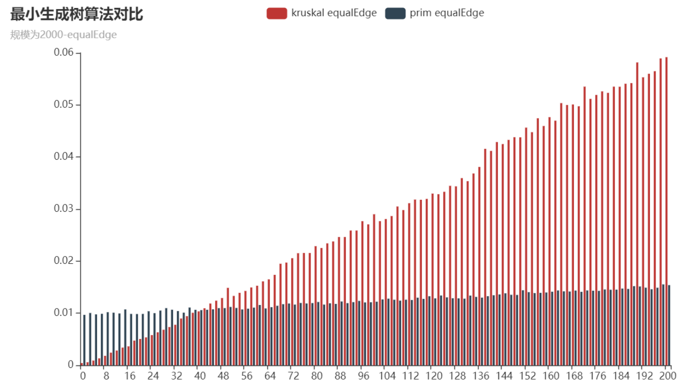


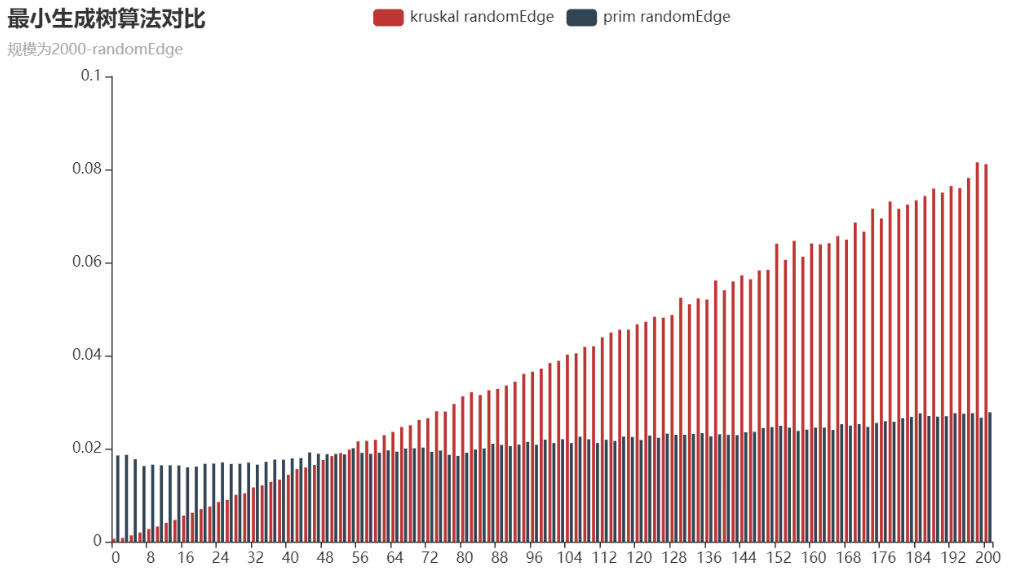


实验过程：扩大范围

- 从之前的研究结果得知，我们应当相对减小疏密度的评定尺度，并将疏密度放在一个较小的区域。
- 右图中我们可以看到，较稀疏的情况下，由边数决定复杂度的Kruskal优势明显。主要由节点数决定的Prim表现稳定

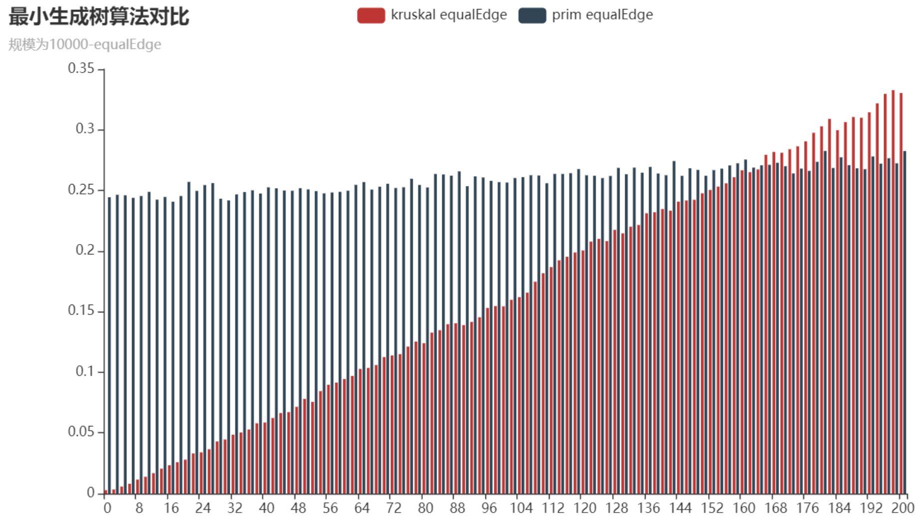


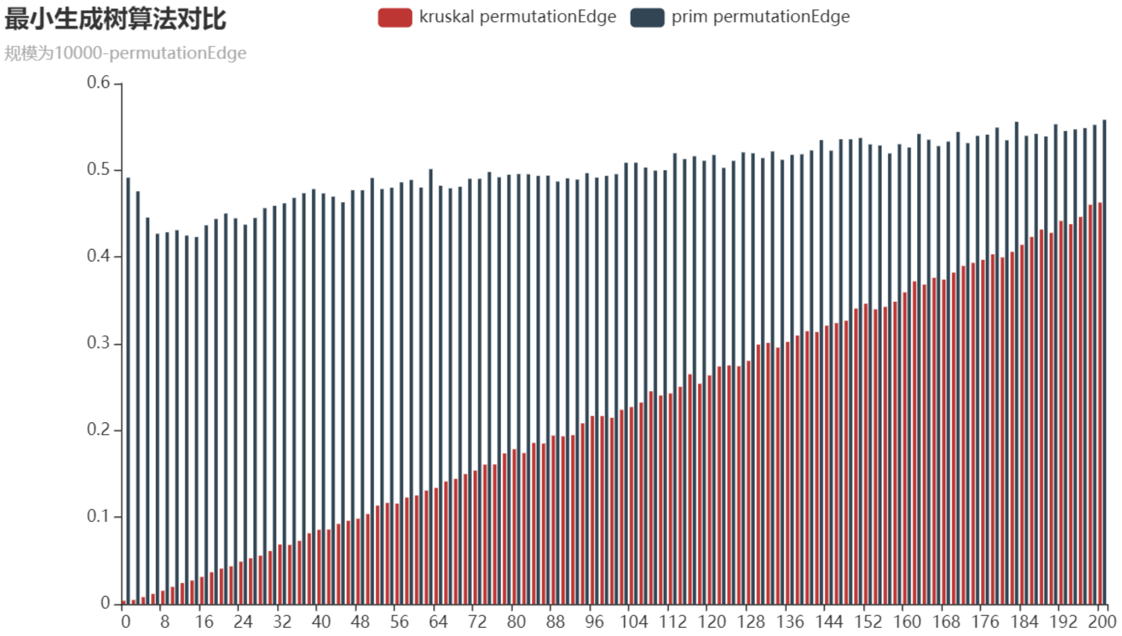


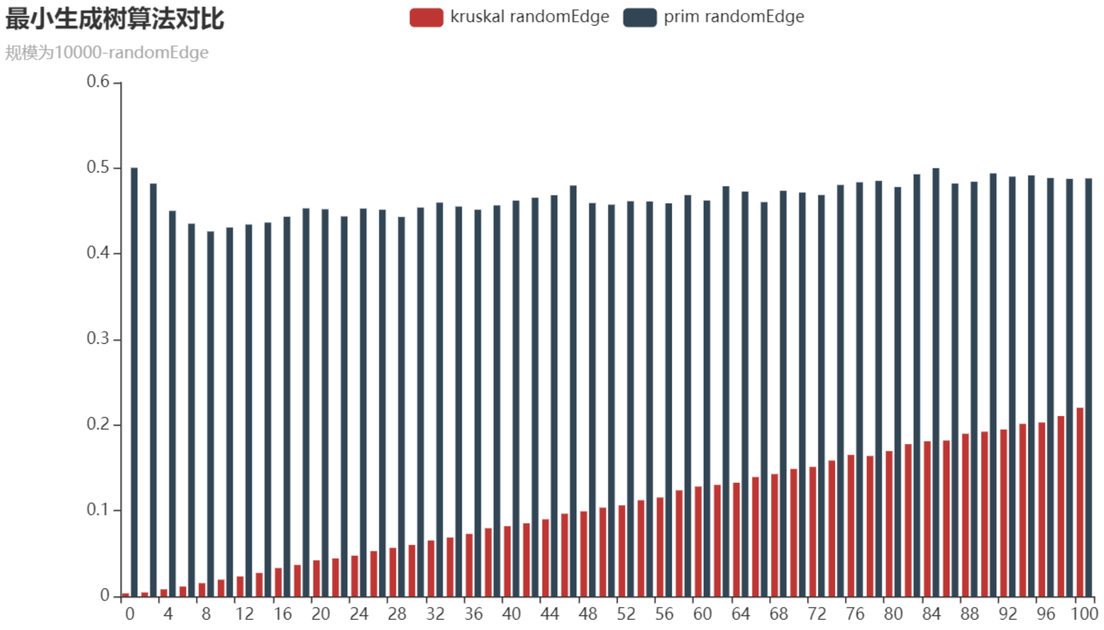


实验过程：继续扩大范围

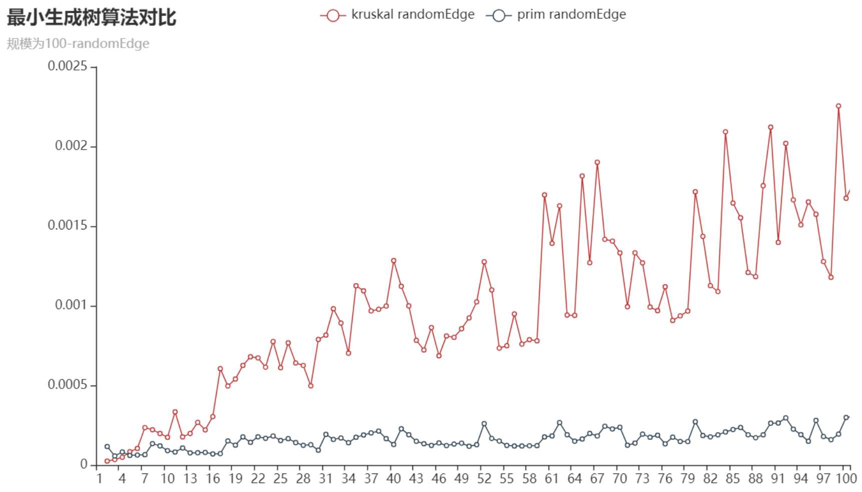
- 我们将实验的范围继续扩大：
- 由于trade off的情形行将消失，不再进行进一步的同类型实验

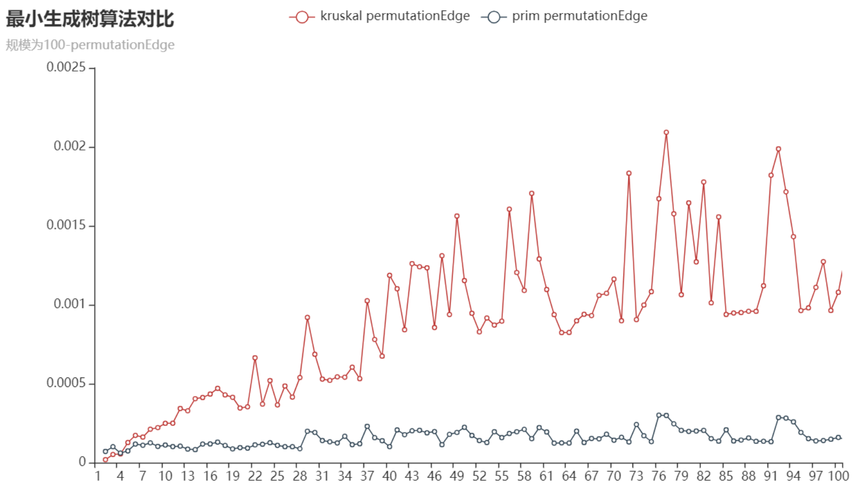


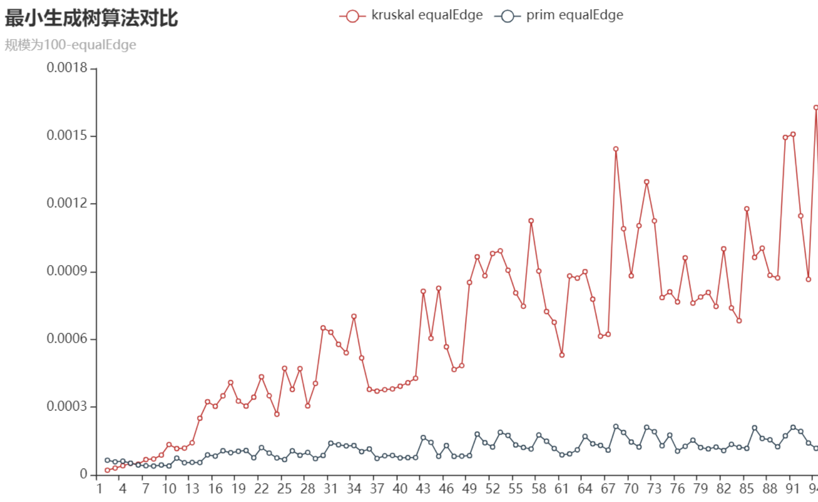


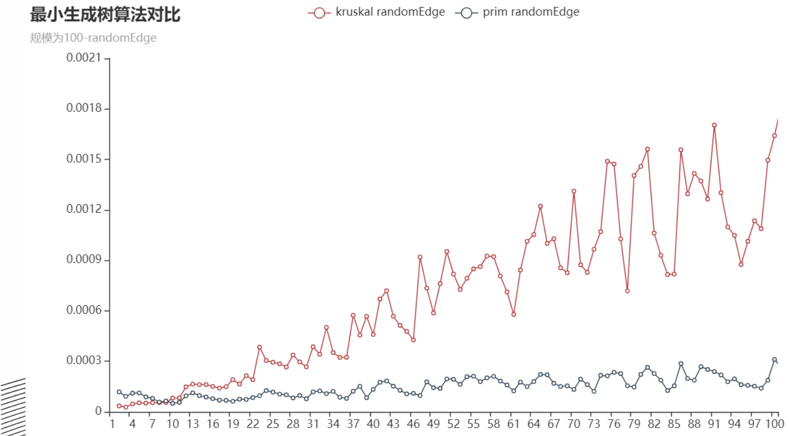


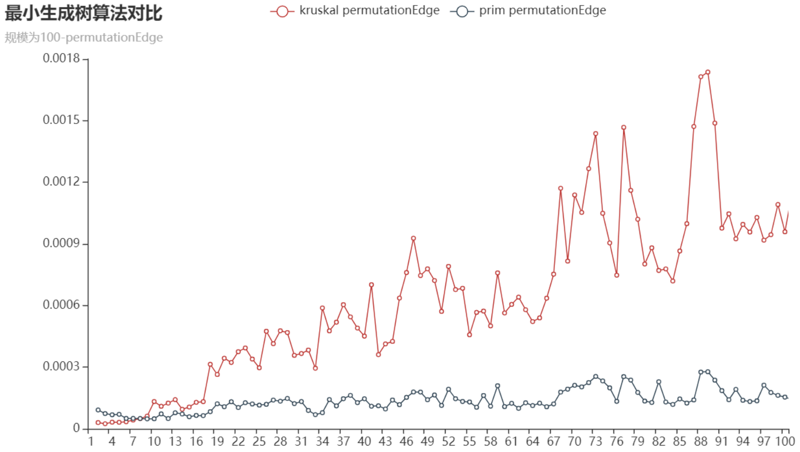
实验过程：添加边的思考

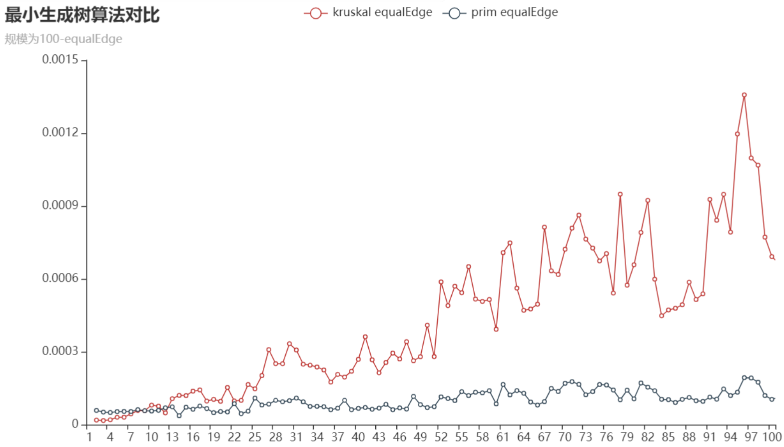

最小生成树算法对比
kruskal randomEdge
prim randomEdge
规模为100-randomEdge
0.0025
0.002
0.0015
0.001
0.0005
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100


最小生成树算法对比
kruskal permutationEdge
prim permutationEdge
规模为100-permutationEdge
0.0025
0.002
0.0015
0.001
0.0005
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100

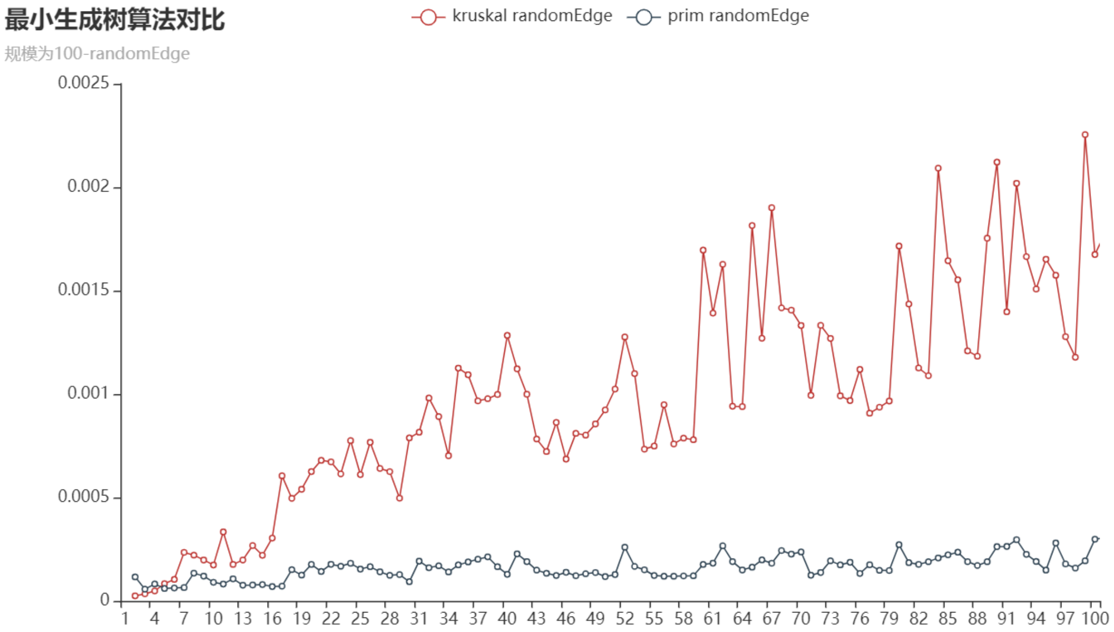

最小生成树算法对比
kruskal equalEdge
prim equalEdge
规模为100-equalEdge
0.0018
0.0015
0.0012
0.0009
0.0006
0.0003
0

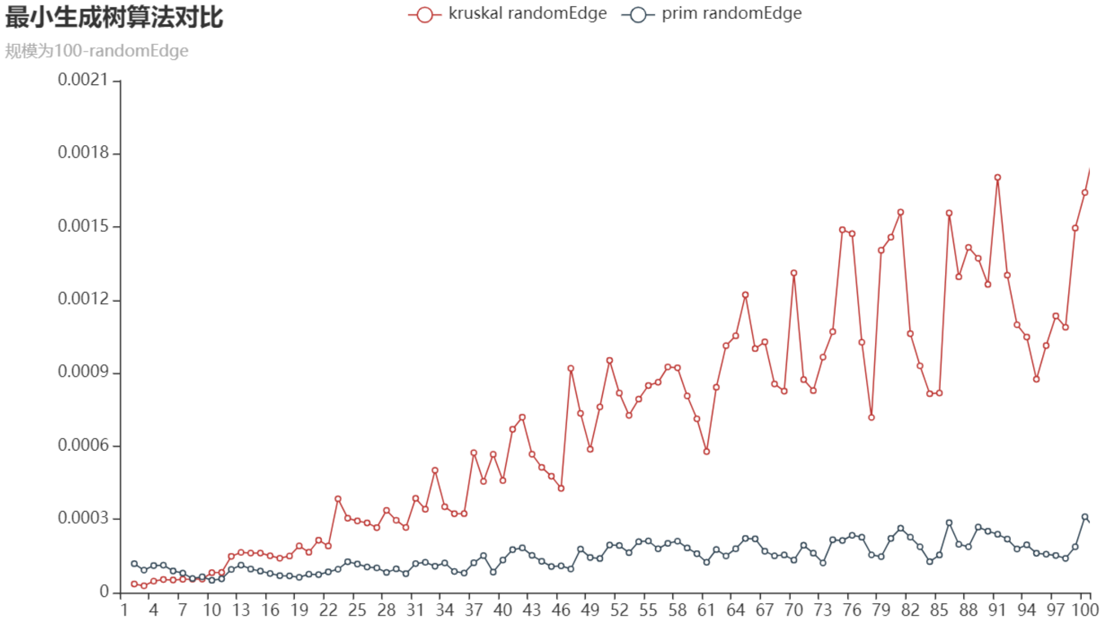

最小生成树算法对比
kruskal randomEdge
prim randomEdge
规模为100-randomEdge
0.0021
0.0018
0.0015
0.0012
0.0009
0.0006
0.0003
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100


最小生成树算法对比
kruskal permutationEdge
prim permutationEdge
规模为100-permutationEdge
0.0018
0.0015
0.0012
0.0009
0.0006
0.0003
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100


最小生成树算法对比
kruskal equalEdge
prim equalEdge
规模为100-equalEdge
0.0015
0.0012
0.0009
0.0006
0.0003
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70 73 76 79 82 85 88 91 94 97 100

问题：星状图带来了什么变化?

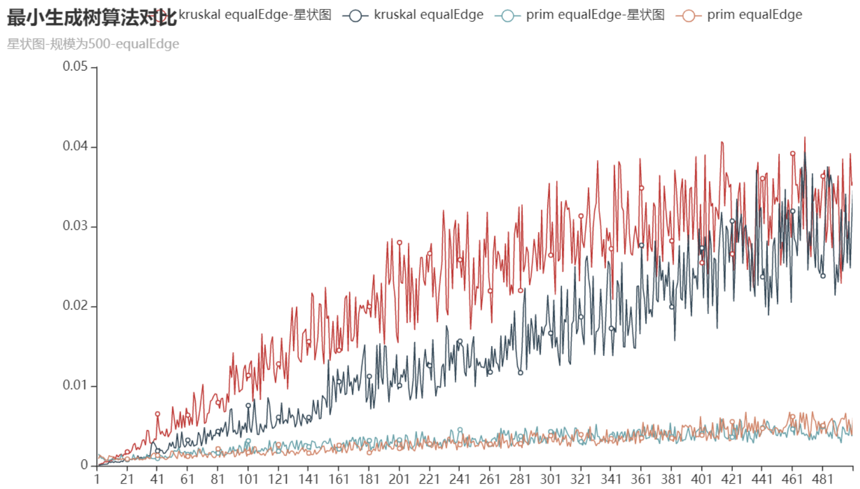


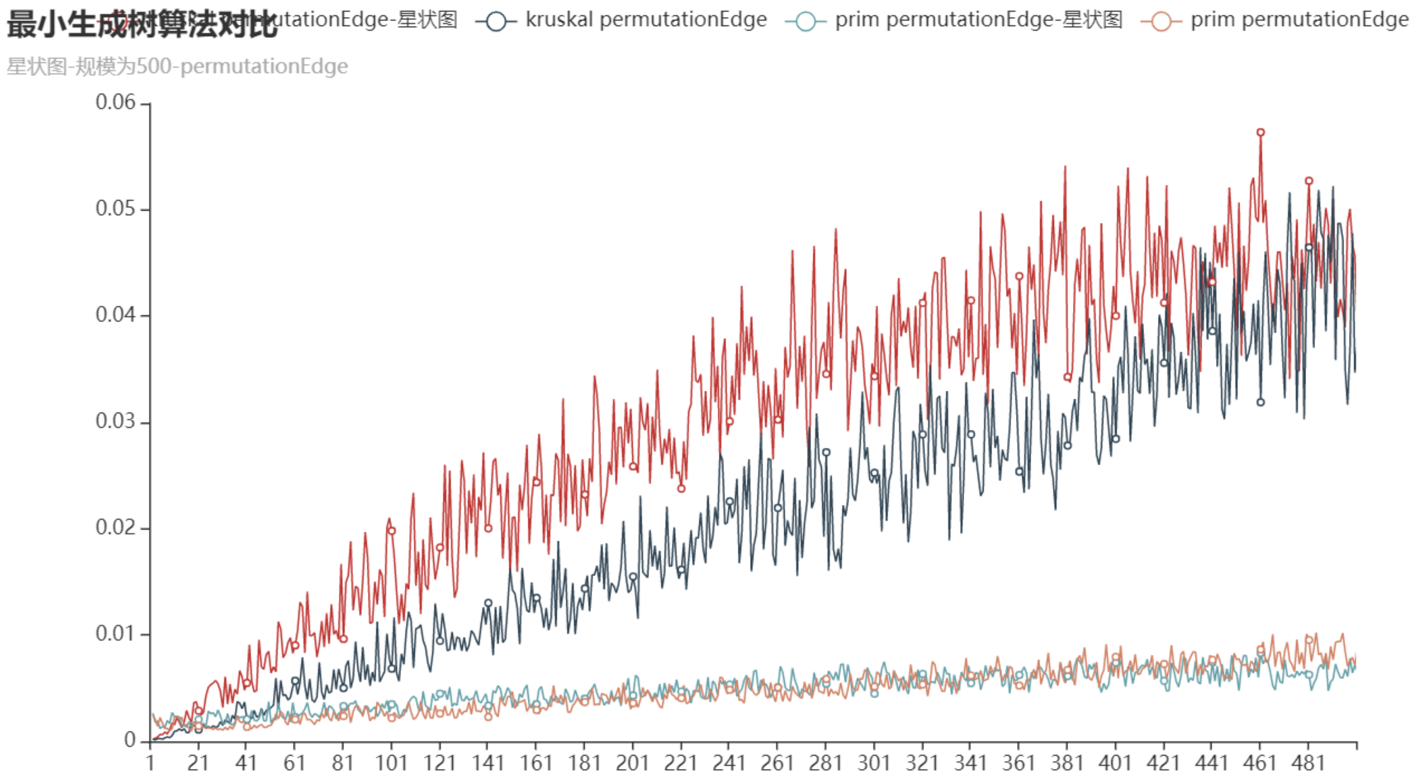


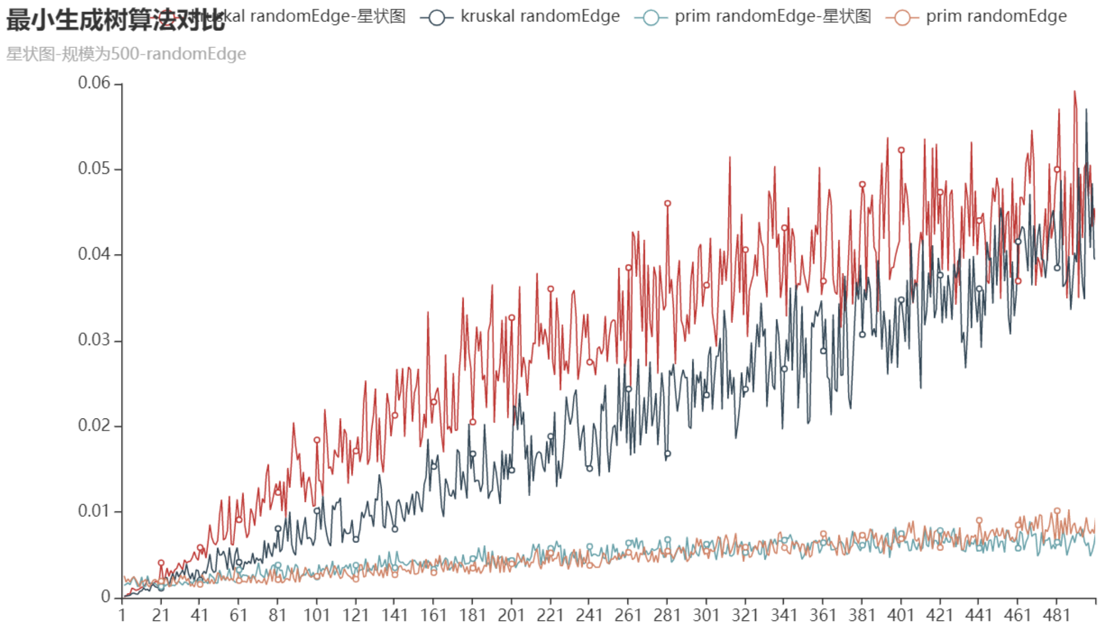
- 我们猜想，分布式选边的Kruskal算法和按点遍历的Prim算法会对不同的图有不同的应对特点。随后我们做验证。

实验过程：星状图

基于如上的猜想，我们构建一系列实验情形，按照单点添边的模式，构建星状图，对比两种MST算法的不同表现。

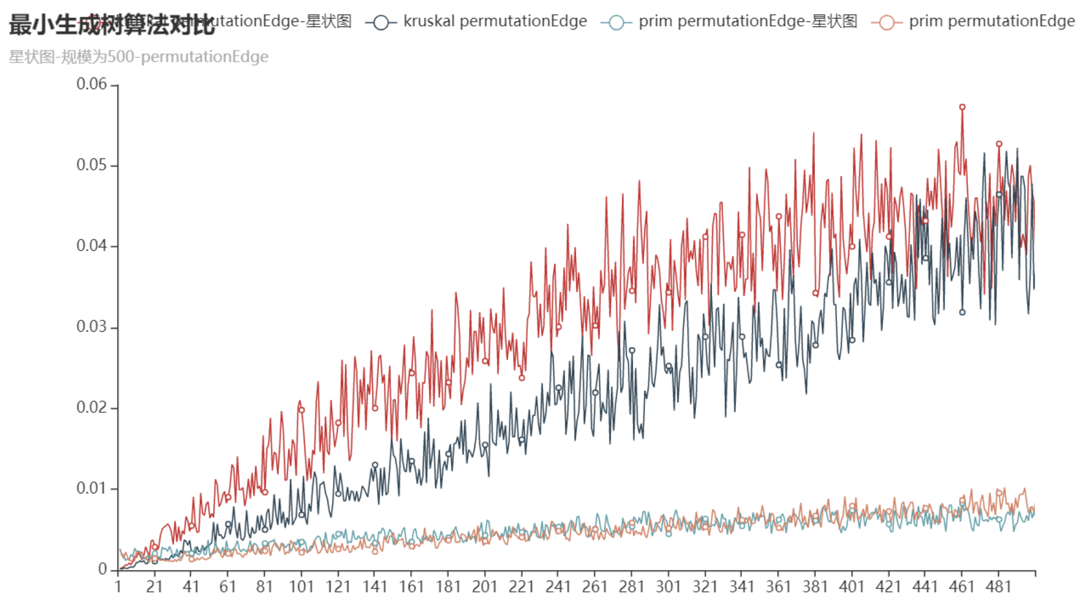






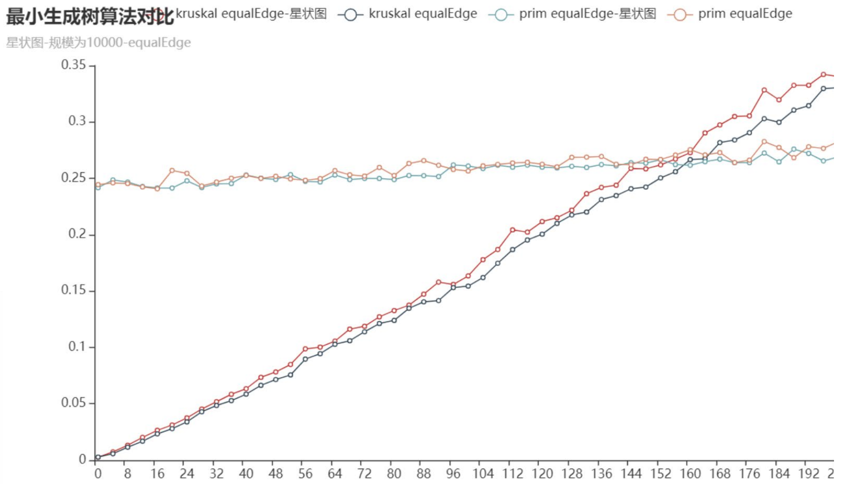
实验过程：星状图的对齐

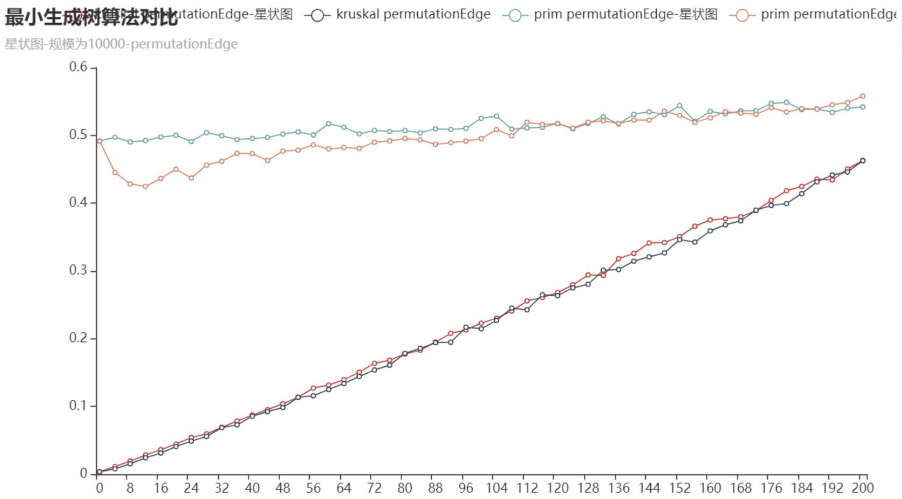
- 分析：由于在Average的实验设计当中，我们每次规律性地添加n/2个点，而星状图为递减等差添加。所以在较低处会出现由于添加点个数不同引发的非控制变量的干扰。
- 在数据较大的时候，添加点数近似为两倍差异。

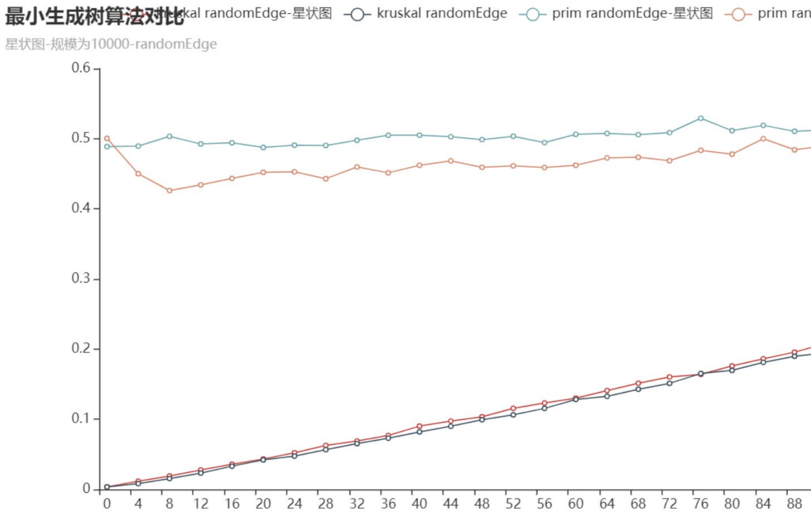


实验过程：星状图的对齐

- 在原有疏密度研究的数据空间中，我们分散取样。从而消除两个数据空间的分布密度的差异。而后得到：



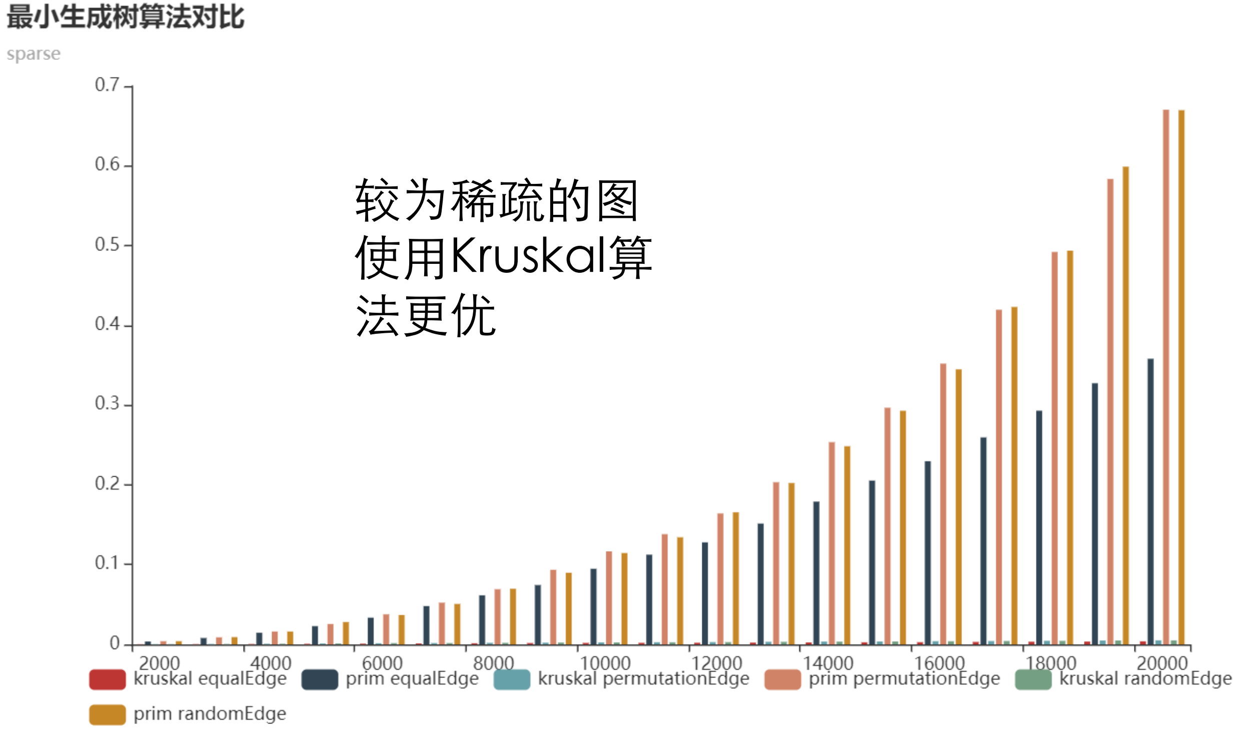




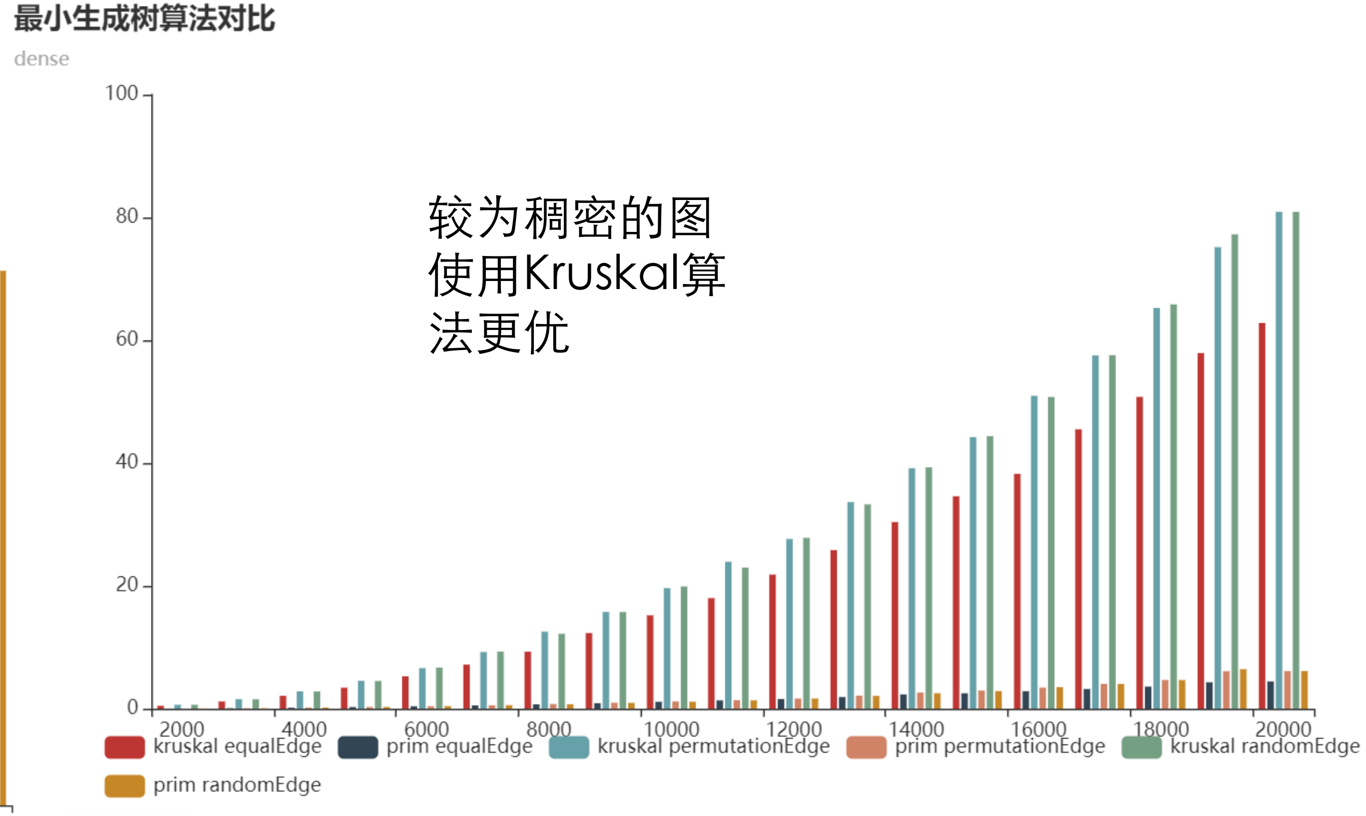
- 这也是容易理解的，Kruskal的时间复杂度瓶颈在于排序。而Prim每个点处理完周边的边之后，其他点扩展仍然会遍历这些边，所以提升并不明显。

实验结果

- 考虑极端稀疏和极端稠密两种情况，展示各自特点



较为稀疏的图使用Kruskal算法更优



较为稠密的图使用Kruskal算法更优

结论和结语

- 在实验中，我们对最小生成树的常用算法进行了对比。考虑到对比的有效性和结果的显著性，我们选择了Kruskal算法和朴素Prim算法，利用三种分配权值的方式构建实验。我们对节点数为100,2000,500,1000,2000,5000,10000下不同疏密度的系列情况进行了实验。
- 另外除了平均赋边的方式，我们构建了星状图的特殊情况，进行了研究。对于这种从节点发散的图，Prim相比于Kruskal并无明显优势。
- 最终确定Kruskal仅在极稀疏的情况下优势明显，但在绝大多数较为稠密的图中，Prim算法都是更优的MST求解法。

Thanks