科技交流与写作课程论文

最小生成树算法的对比分析

Kruskal, Prim 及其堆优化

袁世平*, 范皓年†

2021 年 6 月 16 日

目录

*1700012899, 计算机科学与技术系

†1900012739, 电子信息工程系

1 引言

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。

对于 n 个节点组成的图，所有可能的连接共 $\frac{n(n-1)}{2}$ 条，为构建一个边权最低的连通图，我们需要从中选取 $n-1$ 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”[1]，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”。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。

2 理论分析

2.1 Prim 算法

Prim 算法[3] 是一种基于节点查找的最小生成树算法，基本思想是从一个结点开始，不断加点（而不是 Kruskal 算法的加边）。具体来说，未加入结点中距离已加入结点最小的结点，并用这个结点的邻边更新与其他未加入结点的距离。

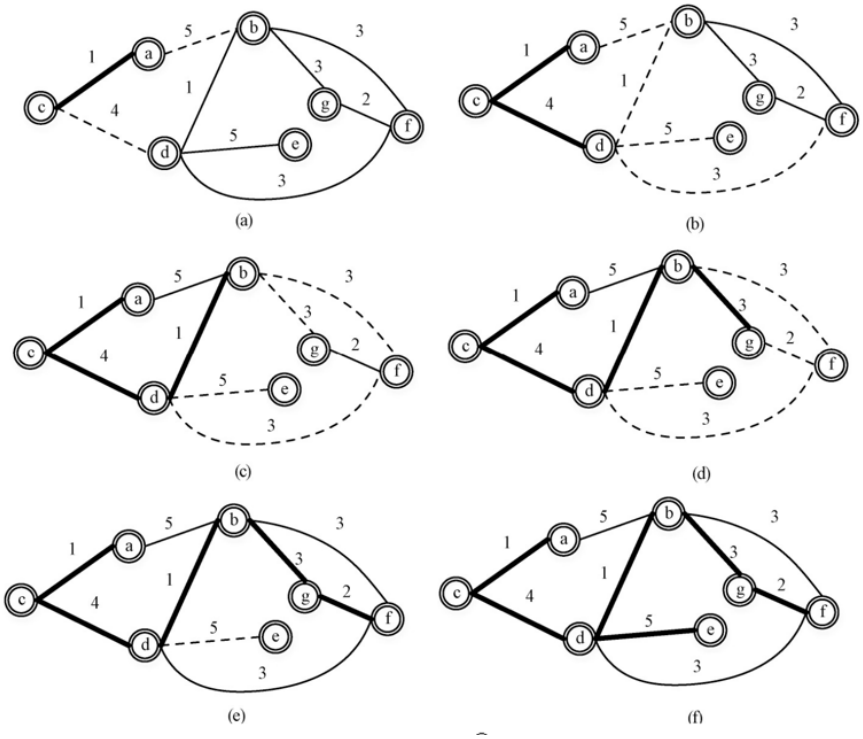


图 1: *Prim 算法的过程示例*

[4]

分析这个过程，在应用 Prim 算法的过程中，每次加入一个点，一共会加入 N 轮，每轮中需要找到一个距离已有点最小的点，使用遍历的方式查找，时间复杂度为 $O(N)$，同时每个点会更新它的邻边，每

```
1   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.
2   Output. The sum of weights of the MST of the input graph.
3   Method.
4   result ← 0
5   choose an arbitrary node in V to be the root
6   dis(root) ← 0
7   for each node v ∈ (V − {root})
8       dis(v) ← ∞
9   rest ← V
10  while rest ≠ ∅
11      cur ← the node with the minimum dis in rest
12      result ← result + dis(cur)
13      rest ← rest − {cur}
14      for each node v ∈ adj(cur)
15          dis(v) ← min(dis(v), g(cur, v))
16  return result
```

图 2: *prim* 算法的伪代码

条边会被两个点更新两次，因此总的所有点更新边的时间复杂度为 $O(M)$。因此 Prim 算法的**时间复杂度**为 $O(N^2 + M)$。

空间上需要存储整个图，**空间复杂度**为 $O(N + M)$。

2.2 Prim+ 堆优化

Prim 堆优化算法是在 Prim 的基础上进行了改进，我们可以看到在 Prim 算法的时间复杂度上，$O(N^2)$ 占据了较大的比例，而它的来源是每一次的查找最大值，因此 Prim 堆优化就是使用堆来找到最大值。每次用边更新节点时，会将如果更新了结点的值，则会将更新的结点的距离加入堆中，因此堆的大小与边数有关，为 $O(M)$。每次更新的时间是 $O(\log M)$ 的，每条边都可能会进行一次，因此**时间复杂度**为 $O(M \log M)$。

而在每次通过堆获得最小值时，时间复杂度为 $O(\log M)$，共进行 N 轮，因此总的时间复杂度为 $O(N \log M + M \log M)$，因为堆的空间复杂度为 $O(M)$，因此**空间复杂度**仍为 $O(N + M)$。

2.3 Kruskal 算法

Kruskal 算法是一种常见并且好写的最小生成树算法，由 Kruskal 发明[2]。该算法的基本思想是从小到大加入边，是一个贪心算法。

算法虽简单，但需要相应的数据结构来支持。具体来说，维护一个森林，查询两个结点是否在同一棵树中，连接两棵树。抽象一点地说，维护一堆集合，查询两个元素是否属于同一集合，合并两个集合。其中，查询两点是否连通和连接两点可以使用并查集维护。

Kruskal 算法的时间复杂度瓶颈在于排序算法，使用快速排序、归并排序等算法进行排序，时间复杂度为 $O(M \log M)$。在之后的并查集中，最坏进行 M 轮判定，每次判定两个结点是否连通需要 $O(\alpha(N))$。就可以得到**时间复杂度**为 $O(M \log M + M\alpha(M, N))$ 的 Kruskal 算法。

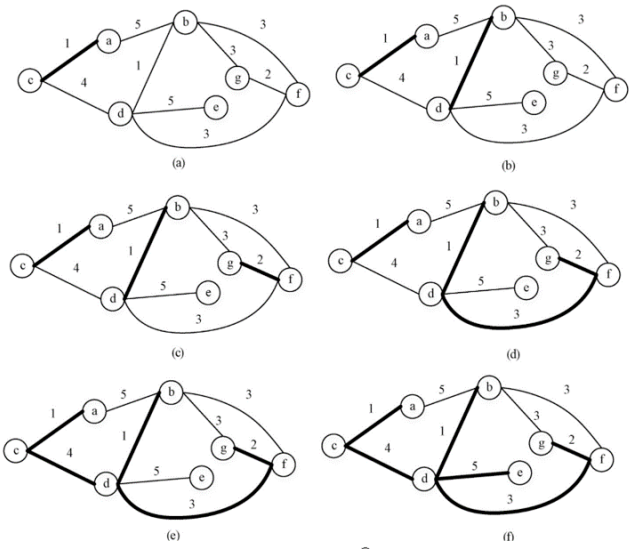


图 3: *Kruskal* 算法实例

伪代码如下：

```
1   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.
2   Output. The edges of the MST of the input graph.
3   Method.
4   result ← ∅
5   sort e into nondecreasing order by weight w
6   for each (u, v, w) in the sorted e
7       if u and v are not connected in the union-find set
8           connect u and v in the union-find set
9           result ← result ⋃ {(u, v, w)}
10  return result
```

图 4: *Kruskal* 算法的伪代码

并查集空间复杂度为 $O(N)$ **空间复杂度**为 $O(N+M)$。

在最小生成树的问题中，通常我们有 $N < M < N^2$ 因此带入关系，我们总结上述算法可以得到表1

其中 Prim 堆优化和 Kruskal 虽然都是 $O(M \log M)$，但是 Prim 堆优化中只有更新了距离值的边才会加入堆中，因此这个 M 的值是会十分小的，尤其是在等权的情况下。而 Kruskal 的 M 则是固定为图中的边数 M，如果使用快速排序，其复杂度甚至可能在 $O(M \log M)$ 到 $O(M^2)$ 间摆动

3 实验设计

为了验证我们在2中给出的分析结论，我们完成如下的实验验证。实验中的计算机为 Ubuntu 16.04.6 LTS (GNU/Linux 4.4.0-142-generic x86_64) 48 核。

在我们的实验中，我们设计了生成边权的程序，生成图的程序，运行实验的脚本，以及实验用的 Kruskal 和 Prim 程序。

	Prim	Prim 堆优化	Kruskal
时间复杂度	$O(N^2)$	$O(M \log M)$	$O(M \log M)$
空间复杂度	$O(M)$	$O(M)$	$O(M)$

表 1: 几种算法的复杂度分析

关于不同算法的评测，我们使用了 C 语言的时间计算库 <sys/time.h> 中的 gettimeofday() 函数计量时间。精确到微秒 ($10^{-6}s$)，忽略读入时间，测试 10 次取平均值。

```
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;
}
```

我们在实验中要验证三类因素的影响，固定节点数情形下稀疏程度的影响，添加边不同方式的影响，边权生成方式的影响。

首先，图的节点个数必然和总耗时是正比的，节点个数越多，总耗时越大。我们对图的分类可以根据节点数的大小进行分类，来验证稀疏程度和加边方式的影响。我们的实验验证从 100 到 1000，以 100 为步长，来实现对于我们工程的正确性的验证。随后我们以 2000 位为步长，从 2000 到 20000 遍历完成更大规模的实验验证。

稀疏程度的验证 从而分析不同稀疏程度对于两个算法的影响。然后我们考虑在 N 个节点的图中，我们考虑边的数量，这里我们比较容易想到两种特殊的图，它们分别代表了稀疏图和稠密图的极端情况：

a.N 个节点的最小连通情况--树

b.N 个节点之间相互都连有边-- 完全图

这两种情形通常出现在 N 较大的情况中，不适合进行分步加边，在这种情况下我们使用的是 vector 进行存储，只有 vector 才能支持 N=20000，M=199990000 的存储。

这种情况下只需要读入边权，然后构建一棵树输出，完全图则只需要两个 for 循环判断两个节点不相同就相连就可以

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

这里构建树的方法非常简单，即对第 i 个节点，随机一个节点 j (j < i)，然后将他们连通，这样就能保证每个节点连通，而且没有环，因为前面的节点不可能有回边。

```
// 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)
```

加边方式 我们想到可以通过加边的形式来从树变成完全图，这里我们根据 Prim 和 Kruskal 算法集中式和分布式地构建其最小生成树的方式，想到了两种方法。第一种是**平均**地在图上加边，每次数目固定。我们从每个点的度数入手实现平均加边，因为最后一个完全图的所有点的度数都为 n-1, 那么在加边的第 T 轮，我们只给度数 <T 的节点之间连边，这样就可以让整个图平均加边，这样只需要进行 n-1 轮，所有的节点就都相互连通。

```
// 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
```

另一种，基于我们对 Prim 算法的思考，我们使用了**集中加边**：由于 Prim 的生成过程是从一个点开始依次寻找最近节点，所以我们猜测以每个点都加满的星状图会对 Prim 有利。我们从每个节点入手，直接让这个节点与所有的节点连边，这样我们执行 n-1 轮，也就让这个图变成完全图了。

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

边权的生成 由于 Kruskal 的算法瓶颈在于排序算法，而排序算法会受到一些边界情况的影响，所以我们构建了如下三种生成方式对新添加的边进行权值分配：

- 等权：直接令所有边的权值为 1
- 随机：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
- 排列：所有边的权值不相同，构造数组，然后调用 <algorithm> 中的 random_shuffle 函数

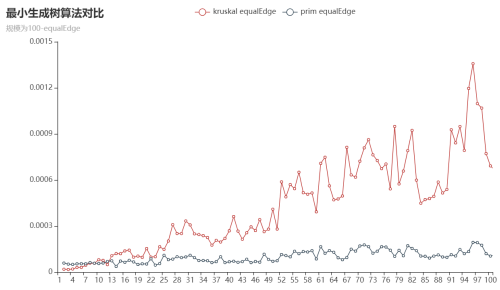
实验中我们编写了测试 shell 脚本，根据设置的节点大小，自动执行我们的测试任务。

```
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
        计算平均值
        枚举进行的轮数
          根据当前的图执行kruskal
        执行平均值
```

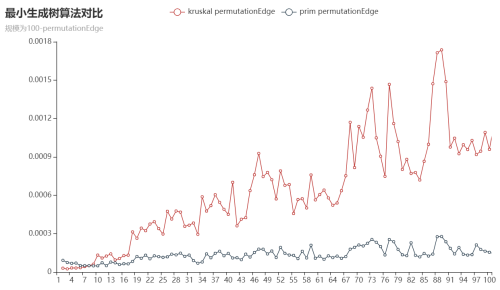
4　实验过程和结果分析

4.1　工程正确性验证

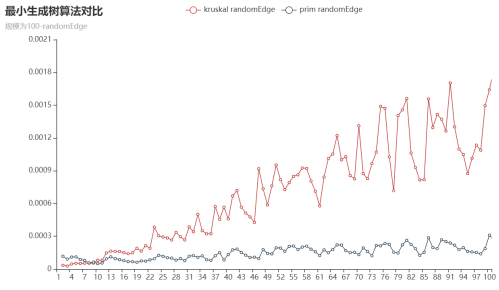
在3中我们已经说到，我们先使用较少的节点进行正确性验证。我们首先测定了 100 节点下，不同权值分配方式对应的稀疏度的影响



(a) 100 节点等权



(b) 100 节点排列权



(c) 100 节点随机权

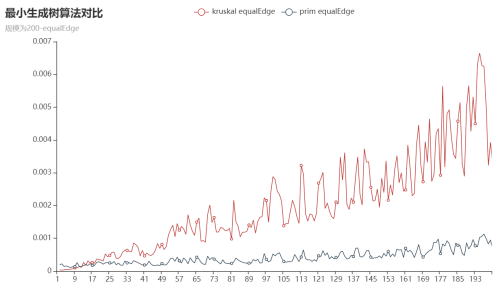
图 5: *100* 节点

实验结果符合预期，即 Kruskal 对于边数敏感，随着边数增多，耗时显著增加。对于 Prim 来说，由于节点数确定，大体上的耗时没有非常显著的变化。

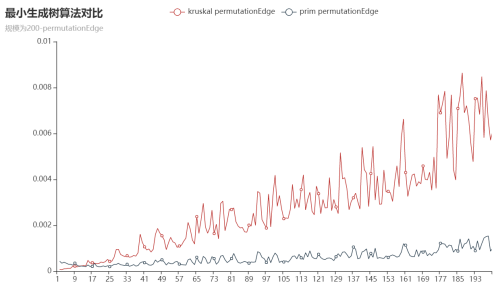
但是我们在这里发现了一个问题，即便是对于 100 这样的少节点数情况，我们添加了少量边之后，仍然表现出稠密图的性质，使得 Kruskal 表现极差。这给我们了一个指导，即在当前的加边方式之下，可以将观察的重点放到较为稀疏的区间上，这样才能较为完整地展现稀疏图 Kruskal 更优的特性。

另外我们发现，三种权值分配方式的耗时依次递增如下：等权、排列权、随机权。这和 Kruskal 中使用到排序有关；另外，这和我们设定的权值大小有关，由于等权的情形中，我们设定所有边全为 1，所以比起另外两种有较大权值的情况，应当在算术、读写等多方面都会更快。

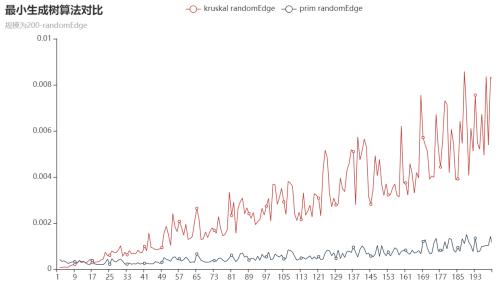
我们在验证部分中还研究了 200 节点和 500 节点的情形。



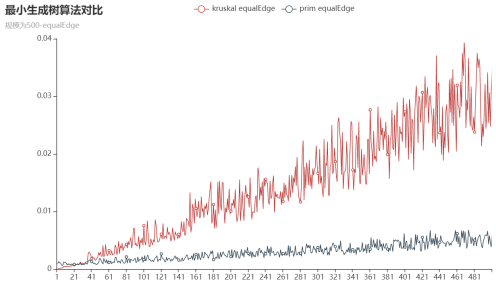
(a) 200 节点等权



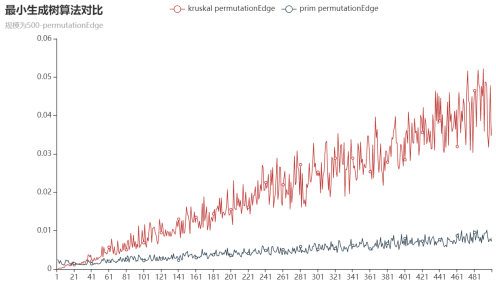
(b) 200 节点排列权



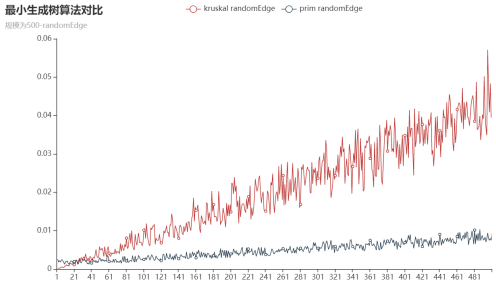
(c) 200 节点随机权



(d) 500 节点等权



(e) 500 节点排列权

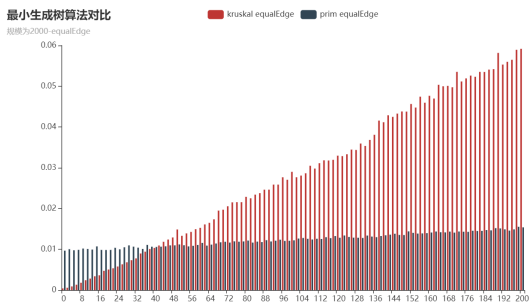


(f) 500 节点随机权

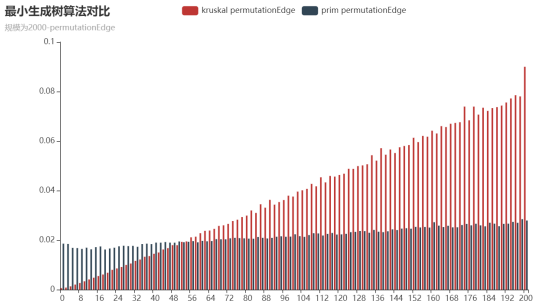
图 6: 更多的验证

4.2 扩大节点数目

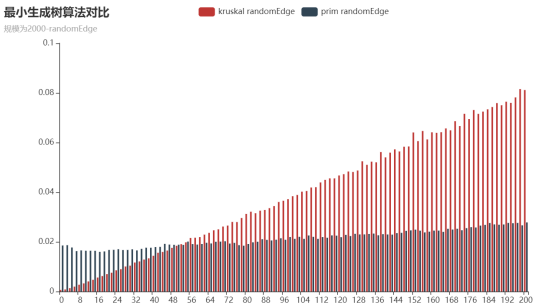
基于之前关于选取观察区间和大概趋势的思考，这里我们对于 2000、5000、10000 三种情形的前 200 次加边的图进行最小生成树算法的对比。结果是显然的，Kruskal 更加适合较为稀疏的情形而 Prim 更加适合较为稠密的情形：



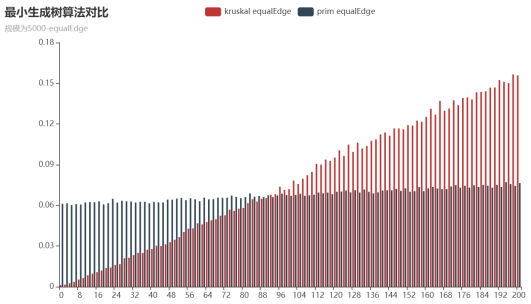
(a) 2000 节点等权



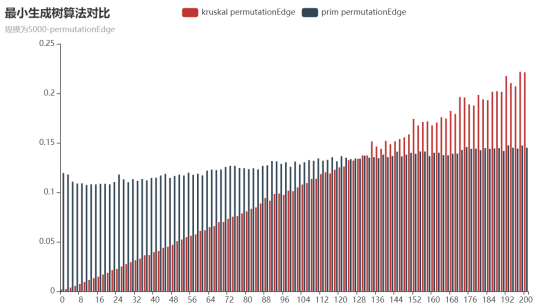
(b) 2000 节点排列权



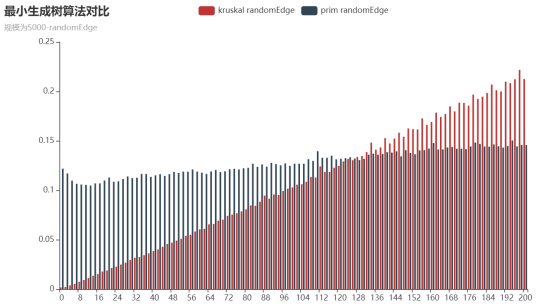
(c) 2000 节点随机权



(d) 5000 节点等权



(e) 5000 节点排列权

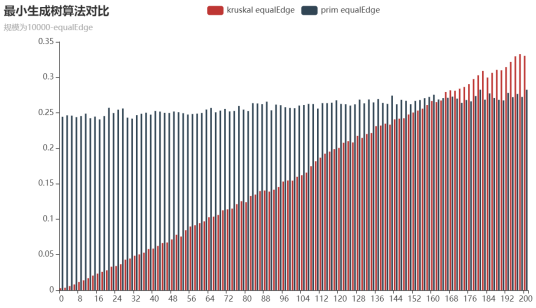


(f) 5000 节点随机权

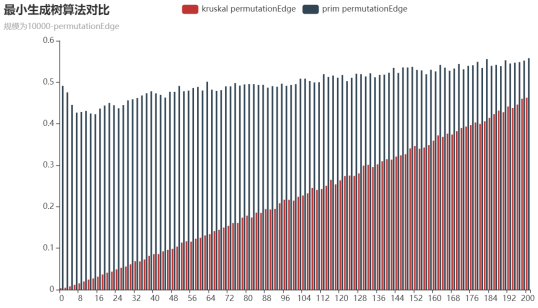
图 7: 较多节点数目情形的验证（*1*）

我们把视角都聚焦在较小的范围，从而可以较为清晰地看到随着边数增多，Prim 算法超过 Kruskal 算法的全过程。另外，这个超越发生的点（我们成为超越点）在三个情形中并不相同，我们还能看出，对于同样加边次数的情形，节点数越高，超越点对应的加边次数越大。

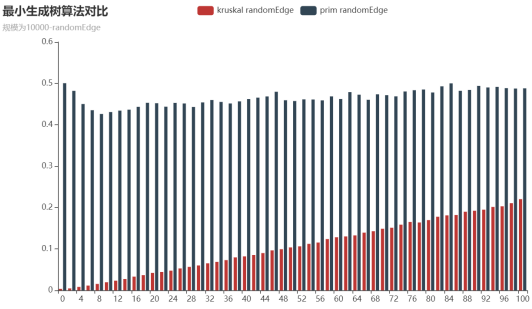
利用表中的数据可以定性地看出这个对应加边次数和节点数正相关，且在较大的节点数下有较好的线性，对于等权的情况，斜率约为 $\frac{1}{50}$：



(a) 10000 节点等权



(b) 10000 节点排列权



(c) 10000 节点随机权

图 8: 较多节点数目情形的验证 (2)

N	500	2000	5000	10000
等权	28	40	94	164
随机权	23	52	126	/
排列权	21	52	128	/
logN	6.21	7.6	8.517	/

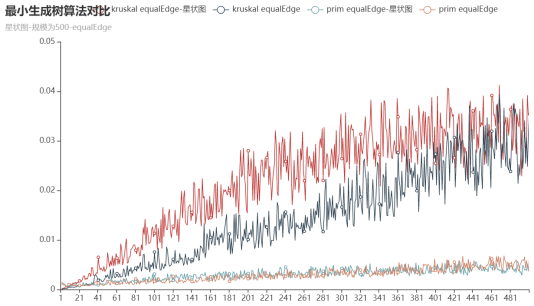
表 2: 不同节点数、不同边权生成方式的超越点位置

4.3 加边方式的对比

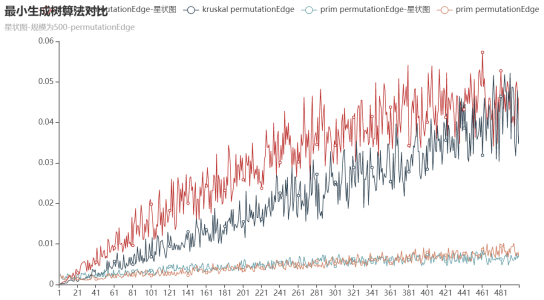
在3中，我们描述了两种加边的方式，在如上的添加边的过程当中，我们作了平均加边。但很自然地，我们有如下的思考：Prim 算法是基于节点向外延展，而 Kruskal 算法是分布式地找边，随后连接节点，所以，将一个个节点依次加满的半满图，可能对于 Prim 算法的表现更加有利。

我们构建一系列实验情形，按照3中给出的单点集中加边的模式，构建星状图，对比两种 MST 算法的不同表现。需要注意的是，在我们的工作和可视化中，星状图、pernode、集中加边的意义是相同的，都是按照点集中加边，每次添加边数为 $n-1, n-2, \cdots, 1, 0$ 依次减少。

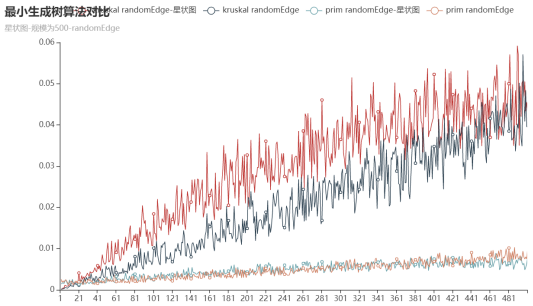
我们验证了几个较少节点的情况，包括 100,200,500，大体结果相同，这里使用 500 节点作为代表的情形进行展示，结果如图9，



(a) 500 节点等权



(b) 500 节点排列权



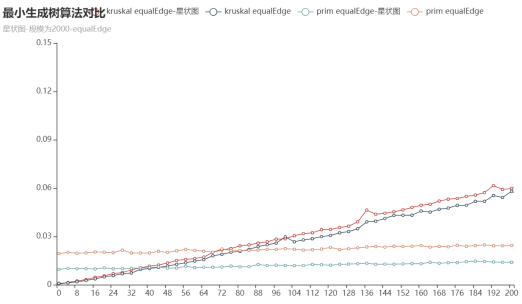
(c) 500 节点随机权

图 9: 不同加边方式的对比

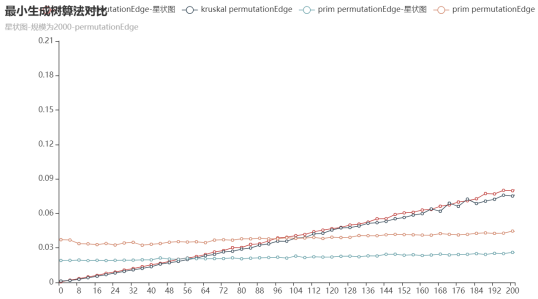
比较看来，Prim 算法表现因为受边数影响较小，所以在两组比较中表现出相对稳定的状态，但由于这种加边方式前期加边数较多，受边数影响较大的 Kruskal 就产生了明显的不同。

考虑到平均加边是每次加 $\frac{n-1}{2}$ 条边，所以这样两种加边方式在初期的加边数有着较大差异，为了消除这种影响，实现控制变量，我们增大节点数，然后在加边的前 200 个情形中，我们近似认为添加边数相差 2 倍，并进行对齐操作，即将加边较多的星状图的情形的数据连续取样而加边较少的情形隔点取样，消除两个数据空间的分布密度的差异，相当于横轴拉伸 2 倍，从而获得的结果如图10和11，耗时就近似只和算法本身对于两种加边的不同适应性有关了。

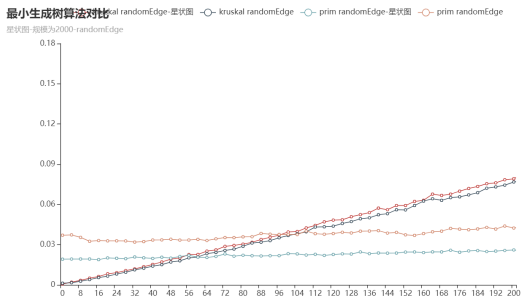
从图10和11中可以看出，对两种算法的影响并不显著，这也是容易理解的，Kruskal 的时间复杂度瓶颈在于排序。而 Prim 每个点处理完周边的边之后，其他点扩展仍然会遍历这些边，所以耗时增加并不明显。



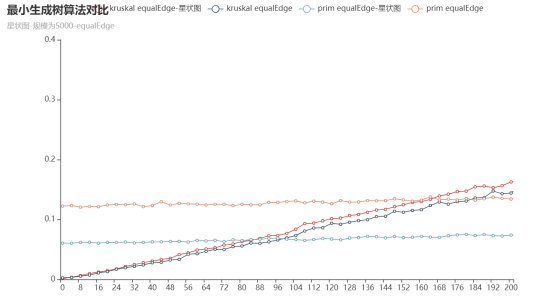
(a) 2000 节点等权



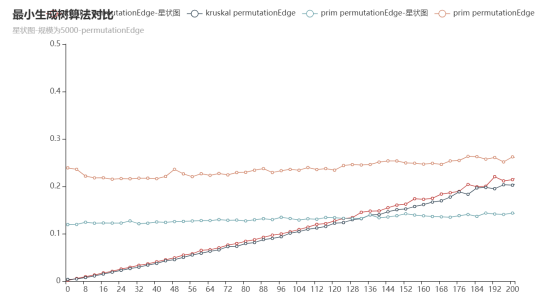
(b) 2000 节点排列权



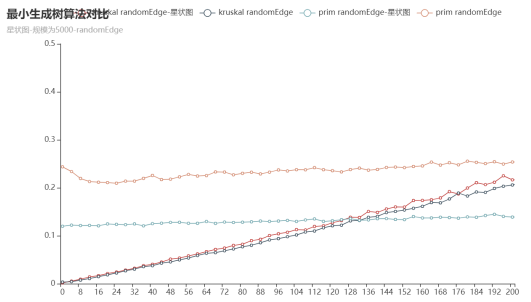
(c) 2000 节点随机权



(d) 5000 节点等权

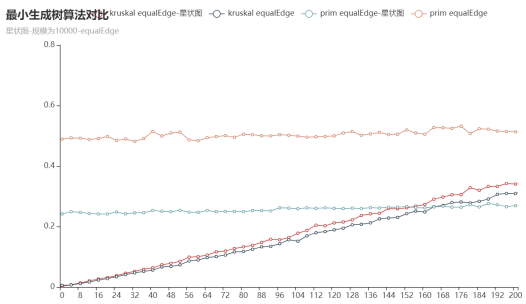


(e) 5000 节点排列权

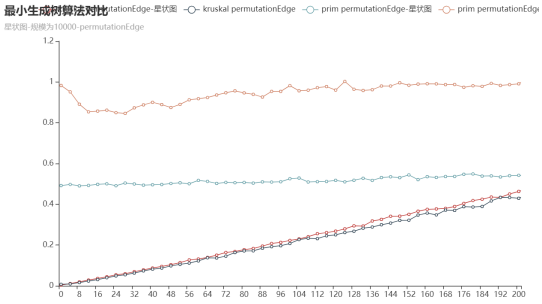


(f) 5000 节点随机权

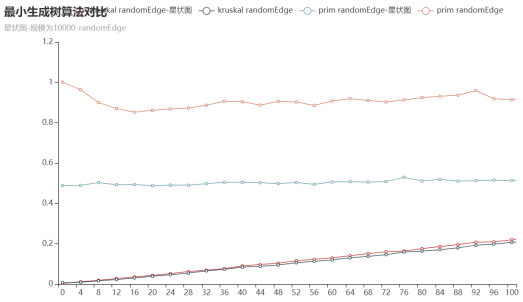
图 10: 两种加边方式对齐之后的对比 *(1)*



(a) 10000 节点等权



(b) 10000 节点排列权

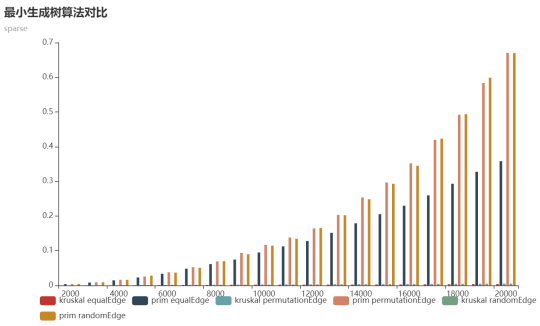


(c) 10000 节点随机权

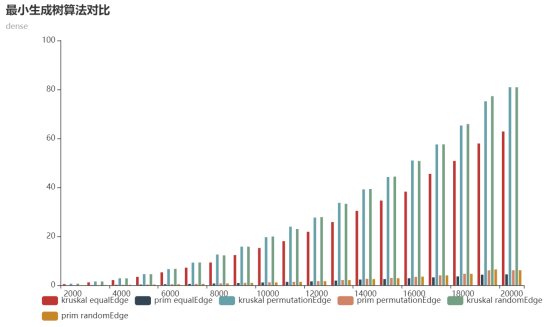
图 11: 两种加边方式对齐之后的对比 *(2)*

4.4 稀疏和稠密的极端情形

算法分析课程中给出的“稠密用 Prim”、“稀疏用 Kruskal”已经经由上述实验完成了多维度验证，我们这里再给出树、完全图两种极稀疏和极稠密的图进行实验验证说明这个经验结论，如图12。



(a) 树



(b) 完全图

图 12: 稀疏适合适合 *Kruskal*，稠密适合 *Prim*

5 结论

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, 200,

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。

参考文献

[1] R. L. Graham and P. Hell. On the history of the minimum spanning tree problem. *Annals of the History of Computing*, 7(1):43–57, 1985.

[2] J. B. Kruskal. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29(1):1–27, 1964.

[3] R. C. Prim. Shortest connection networks and some generalizations. *The Bell System Technical Journal*, 36(6):1389–1401, 1957.

[4] 贺军忠 and 王丽君. Kruskal 和 prim 算法的分析研究与比较. 陇东学院学报, 31(2):8–11, 2020.