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”。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。

2 算法综述

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，每次要选择距离最小的一个结点，以及用新的边更新其他结点的距离。

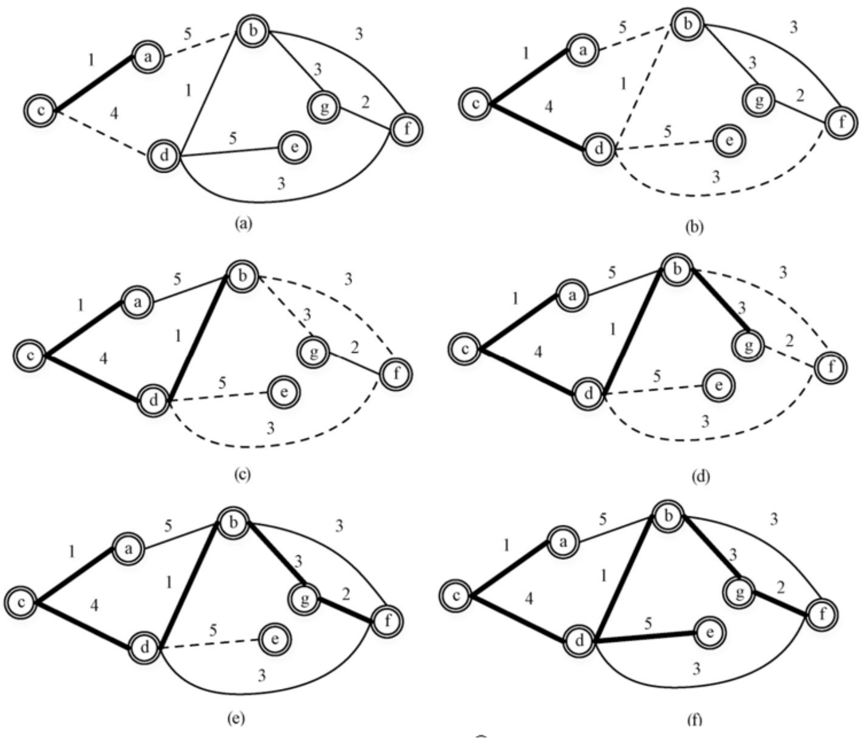


图 1: *Prim* 算法的过程示例

[4]

范皓年

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*

图 2: *prim* 算法的伪代码

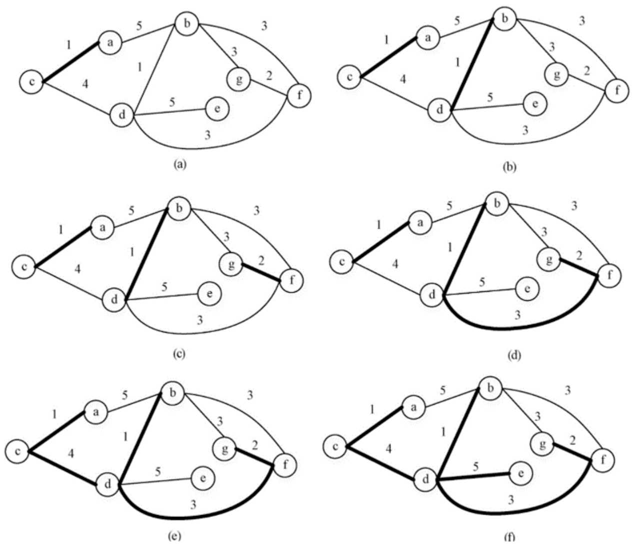


图 3: *Kruskal* 算法实例

2.2 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，如果使用 $O(m\log m)$ 的排序算法，并且使用 $O(m\alpha(m,n))$ 或者 $O(m\log n)$ 的并查集，就可以得到时间复杂度为 $m\log m$ 的 Kruskal 算法。

伪代码如下：

```
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 ← ∅
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 ← result ⋃ {(u, v, w)}
return result
```

图 4: *Kruskal 算法的伪代码*

3 理论分析

4 实验设计和实验过程

4.1 实验的总体设计

为了验证我们在3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。

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。

```c
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。我们在不同的节点下进行实验，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。

对于稀疏程度的验证：我们从一个树开始，一直加满到完全图，从而分析不同稀疏程度对于两个算法的影响。

构图的方式我们还要给出分类说明。构建树的过程如下：

```
// 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)
```

构建完全图的过程如下：

```
// 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)
```

构建树（空图）和完全图之间的半满图，我们采用两种加边方式，这个也是我们所需要探究的一项内容。由于 Prim 的找点方法是从一个点开始轮替寻找最近节点，所以我们猜测以一点发散开的星状图会对 Prim 有利。于是我们除了尝试平均加边以外，还尝试了集中加边、单点加满的方式，逐渐获得完全图。

```
// 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
```

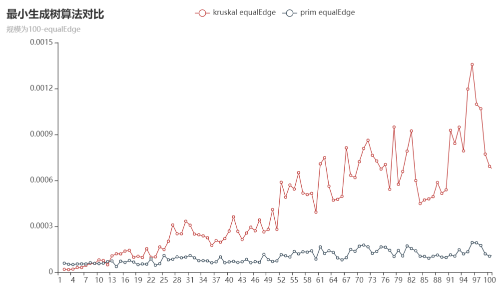
```
    u = T
    for v from 1 to n
        //找到所有u没有连接的节点都连接
        if u != v and !connect(u,v)
            add_edge(u, v)
```

关于边权的生成：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：

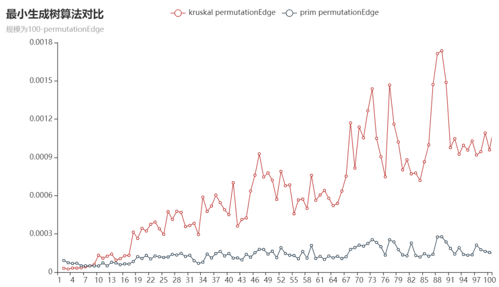
- 等权：直接令所有边的权值为 1
- 随机：所有边的权值随机，使用 <cstdlib> 库中的伪随机数获取 (1,m) 之间的随机数
- 排列：所有边的权值不相同，构造数组，然后调用 <algorithm> 中的 random_shuffle 函数

4.2 工程正确性验证

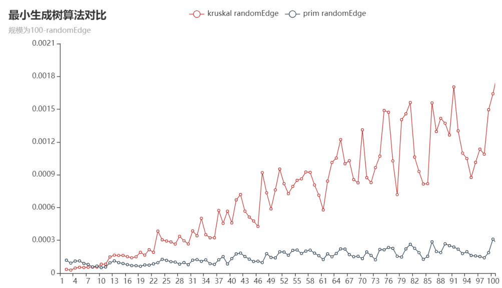
在4.1中我们已经说到，我们先使用较少的节点进行正确性验证。我们首先测定了 100 节点下，不同权值分配方式对应的稀疏度的影响



(a) 100 节点等权



(b) 100 节点排列权



(c) 100 节点随机权

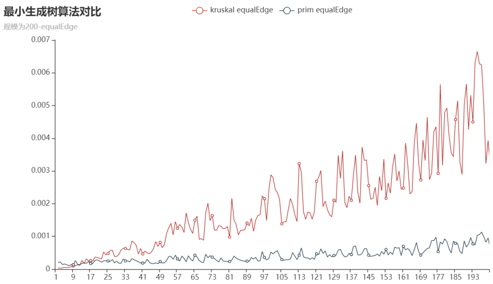
图 5: *100 节点*

我们发现，大体上满足我们的实验分析，即 Kruskal 对于边的多少比较敏感。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。

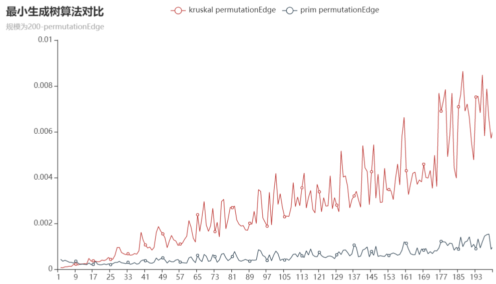
但是我们在这里发现了一个问题，即便是对于 100 这样的少节点数情况，我们添加了少量边之后，仍然表现出稠密图的性质，使得 Kruskal 表现极差。这给我们了一个指导，即在当前的加边方式之下，可以将观察的重点放到较为稀疏的区间上，这样才能较为完整地展现稀疏图 Kruskal 更优的特性。

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。这是和 Kruskal 中使用到排序有关。

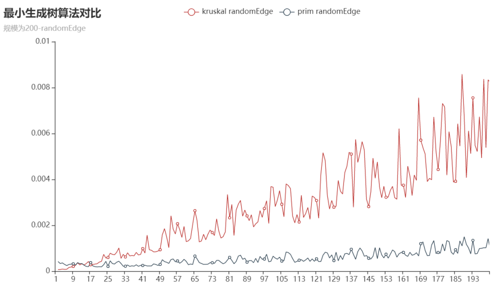
我们在验证部分中还研究了 200 节点和 500 节点的情形。



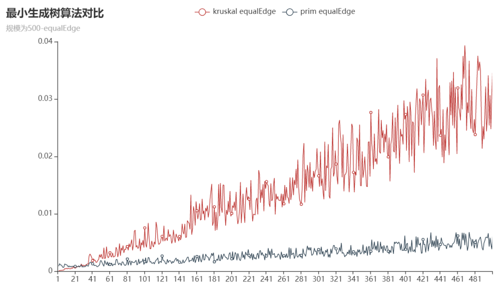
(a) 200 节点等权



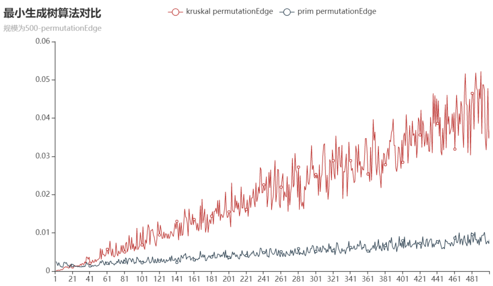
(b) 200 节点排列权



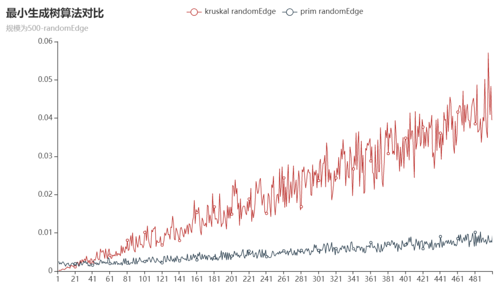
(c) 200 节点随机权



(d) 500 节点等权



(e) 500 节点排列权



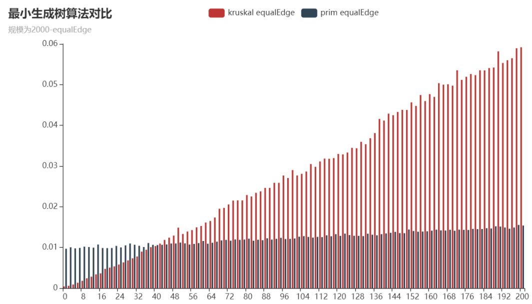
(f) 500 节点随机权

图 6: 更多的验证

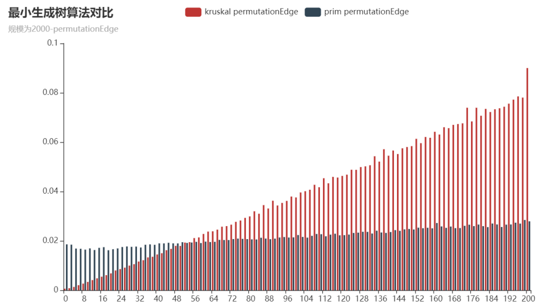
4.3 扩大节点数目

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：

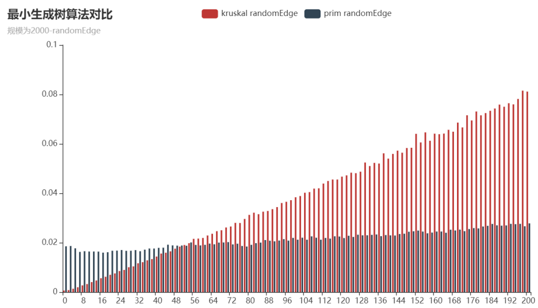
我们把视角都聚焦在较小的范围，从而可以较为清晰地看到随着边数增多，Prim 算法超过 Kruskal 算法的全过程。另外，这个超越发生的点（我们成为超越点）在三个情形中并不相同，我们还能看出，对于同样加边次数的情形，节点数越高，超越点对应的加边次数越大。且定性地看出这个对应加边次数和节点数正相关。



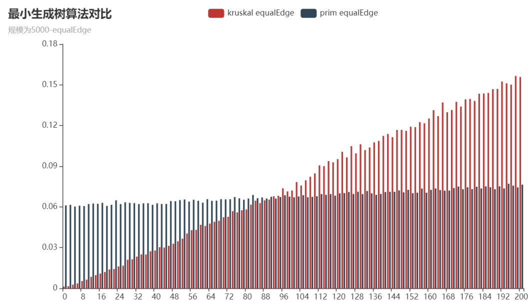
(a) 2000 节点等权



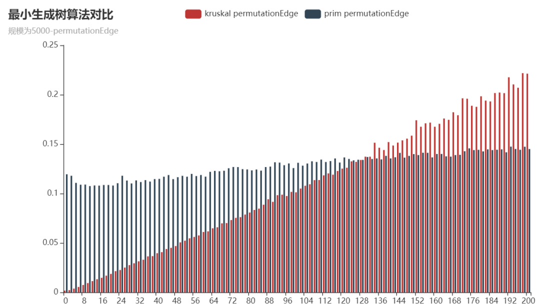
(b) 2000 节点排列权



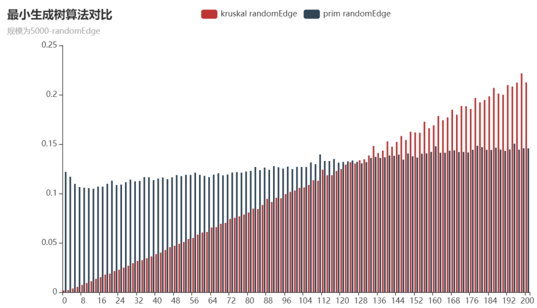
(c) 2000 节点随机权



(d) 5000 节点等权

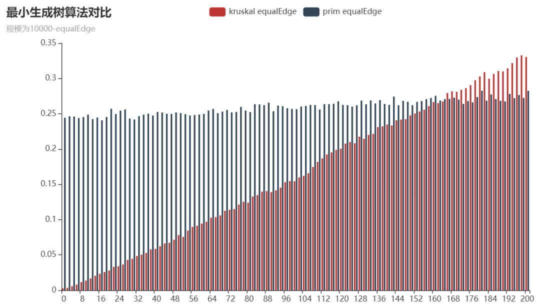


(e) 5000 节点排列权

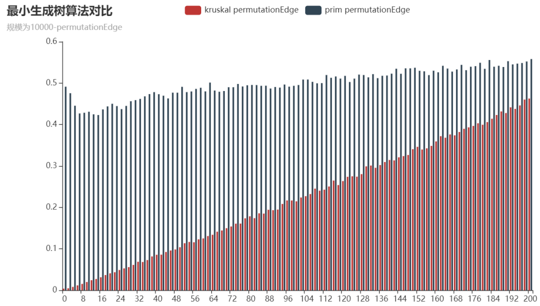


(f) 5000 节点随机权

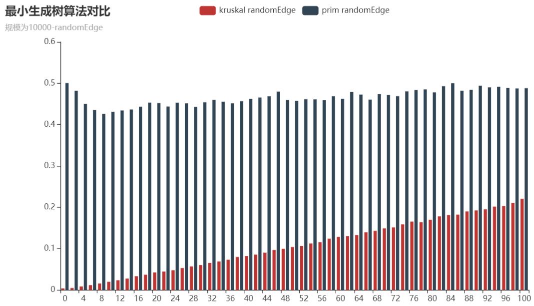
图 7: 较多节点数目情形的验证（*1*）



(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）

5 实验结果分析

6 结论

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.