华中科技大学计算机科学与技术学院

算法分析与设计报告

专　　业：	计算机科学与技术
班　　级：	计算机 ACM1701 班
学　　号：	U201714780
姓　　名：	刘晨彦
成　　绩：	
指导教师：	何琨

完成日期： 2019 年 11 月 15 日

实验一

一、实验题目：最小生成树（MST）算法的实现

二、实验目的与内容

1、实验目的：

本次实验的目的是：了解多种最小生成数独的算法，掌握这些算法的具体实现方法，分析算法复杂度，学习算法的优化方法。

2、实验内容：

两人一组，不限编程语言，各自实现最小生成树的算法，分析复杂度、证明算法的正确性，共同完成 PowerPoint 进行展示交流。

三、算法设计

1、Kruskal 算法描述

程序描述：

STEP 1: 以(v_i, v_j, w_{ij})的形式输入图$G=(V,E), v_i, v_j \in V, (v_i, v_j) \in E, w_{ij}$是$(v_i, v_j)$的权重。初始化 MST 的边集$T=\emptyset$。

STEP 2: 根据输入的边权重进行从小到大的排序。

STEP 3: 选择当前 E 中未选择过的权重最小边e，若 E 中没有未选择过的边则至 STEP6。

STEP 4: 根据 Find()判断e的两点是否已经连通。若不连通，至 STEP5；若连通，至 STEP2。

STEP 5: 将边e加入边集 T，回到 STEP2。

STEP 6：输出 T

Find 函数描述：

STEP 1：调用 root_node()函数判断两个点的根节点是否一致，若一致则返回 False。

STEP 2：比较两个节点的根节点大小，将大的根节点成为小的根节点的子节点。

STEP 3：返回 True

Root_node()函数描述：

STEP 1：若节点的父节点为自身，则返回该节点。

STEP 2：否则递归调用 Root_node()函数，将当前节点的父节点作为参数送入。

2、Kruskal 算法流程图

算法描述的流程图如图 1.1：

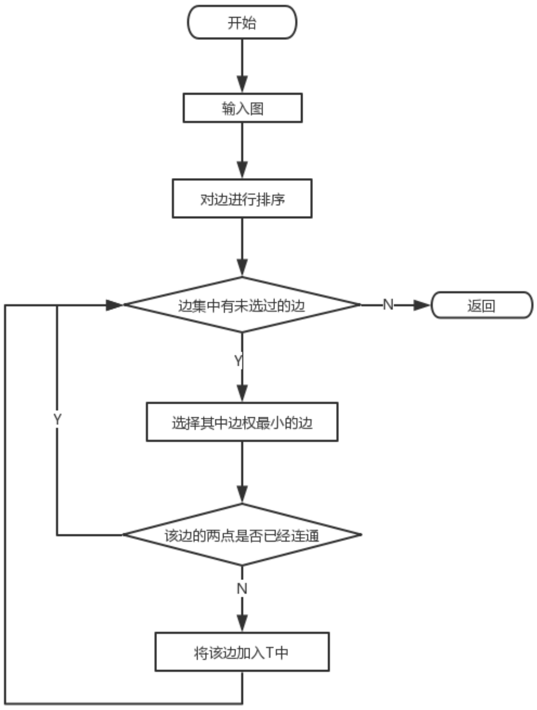

图 1.1 Kruskal 算法流程图

3、Prim 算法描述

程序描述：

STEP 1: 以 (v_i, v_j, w_{ij})的形式输入图$G = (V, E), v_i, v_j \in V, (v_i, v_j) \in E, w_{ij}$是$(v_i, v_j)$的权重。初始化 MST 的边集$T = \emptyset$，点集$X = \{v\}$。$v$为$V$中任意一点。

STEP 2: 遍历边集E，选择一条边e，权重最小且只有一个点在点集X中。若找不到此边则至 STEP4。

STEP 3: 将边e加入边集T，将边中的不在X中的点加入X中。回到 STEP2。

STEP 4: 输出 T。

4、Prim 算法流程图

算法描述的流程图如图 1.2：

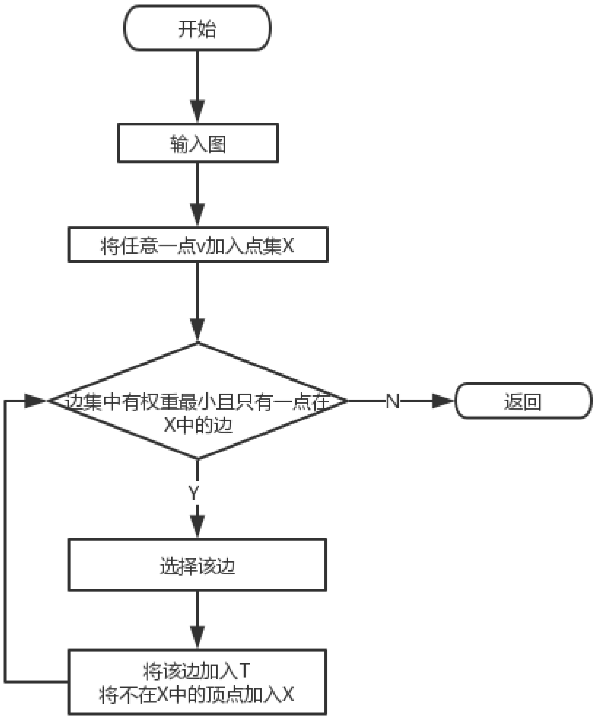

图 1.2 Kruskal 算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Sublime Text3
语言版本：python3.7.4

五、实验过程

Kruskal 算法的 MST 程序代码如下：

Kruskal 算法：

```
def Kruskal(Edges_and_Weights, Num_of_Nodes):
    Nodes_and_Group = {}
    TotalWeight = 0
    EdgeSet = []
    Edges_and_Weights = sorted(Edges_and_Weights, key = lambda x: x[2])

    for i in range(0,Num_of_Nodes):
        Nodes_and_Group[i] = i

    for Edge_and_Weight in Edges_and_Weights:
        if find(Edge_and_Weight[0], Edge_and_Weight[1], Nodes_and_Group):
```

```
                #if 2 nodes are already connected
                EdgeSet.append((Edge_and_Weight[0], Edge_and_Weight[1]))#add
this edge to MST
                TotalWeight += Edge_and_Weight[2]#add the weight to the MST
    return [EdgeSet,TotalWeight]
```

Find 函数

```
def find(i, j, Nodes_and_Group):
    wait_to_change = 0
    goal_of_change = 0

    if root_node(Nodes_and_Group,i) == root_node(Nodes_and_Group,j):
        #if node i and j are already connected:
        return False
    else:
        #change the group number to the smaller one
        if root_node(Nodes_and_Group,i) < root_node(Nodes_and_Group,j):
            Nodes_and_Group[root_node(Nodes_and_Group,j)] =
root_node(Nodes_and_Group,i)
        else:
            Nodes_and_Group[root_node(Nodes_and_Group,i)] =
root_node(Nodes_and_Group,j)

    return True
```

Root node 函数：

```
def root_node(Nodes_and_Group, i):
    if Nodes_and_Group[i] == i:
        return i
    else:
        return root_node(Nodes_and_Group, Nodes_and_Group[i])
```

Prim 算法的 MST 程序代码如下：

Prim 算法：

```
def Prim(Edges_and_Weights, Num_of_Nodes):
    Nodes_in_MST = [0]
    EdgeSet = []
    TotalWeight = 0

    while len(EdgeSet) < Num_of_Nodes - 1:
        MinEdgeWeight = 65535
        for Edge_and_Weight in Edges_and_Weights:
            if ((Edge_and_Weight[0] not in Nodes_in_MST and
Edge_and_Weight[1] in Nodes_in_MST) or\
                        (Edge_and_Weight[0] in Nodes_in_MST and
Edge_and_Weight[1] not in Nodes_in_MST) )and\
```

```
                Edge_and_Weight[2] < MinEdgeWeight:
                    MinEdgeWeight = Edge_and_Weight[2]
                    CandidatePair = (Edge_and_Weight[0], Edge_and_Weight[1])

            EdgeSet.append(CandidatePair)
            TotalWeight += MinEdgeWeight
            if CandidatePair[0] not in Nodes_in_MST:
                #check which node not in Nodes_in_MST, and add this node into
Nodes_in_MST
                Nodes_in_MST.append(CandidatePair[0])
            else:
                Nodes_in_MST.append(CandidatePair[1])
        return [EdgeSet,TotalWeight]
```

六、算法测试

Kruskal 算法测试样例 1

样例输入	设计理由	理论输出	样例输出
[(0,1,3),(0,2,9),(0,5,6),(1,2,9),(1,3,9),(1,4,2),(1,5,4),(2,3,8),(2,9,18),(3,4,8),(3,6,7),(3,8,9),(3,9,10),(4,5,2),(4,6,9),(5,6,9),(6,7,4),(6,8,5),(7,8,1),(7,9,4),(8,9,3)]	检查算法是否正确	Kruskal Algorithm: The edges of the MST: [(7, 8), (1, 4), (4, 5), (0, 1), (8, 9), (6, 7), (3, 6), (2, 3), (3, 4)] Sum of the weight of MST: 38	Kruskal Algorithm: The edges of the MST: [(7, 8), (1, 4), (4, 5), (0, 1), (8, 9), (6, 7), (3, 6), (2, 3), (3, 4)] Sum of the weight of MST: 38 [Finished in 0.4s]

由于理论输出与样例输出相符，所以 Kruskal 算法测试样例 1 验证成功。

Kruskal 算法测试样例 2

样例输入	设计理由	理论输出	样例输出
（随机生成的有 500 个定点和 1000 条边的权重随机的图）	检查算法用时	较快的时间内能够完成	Time: kru: 2 ms [Finished in 0.3s]

由于理论输出与期望相符，所以 Kruskal 算法测试样例 2 验证成功。

综上，Kruskal 算法通过所有样例的测试。

Prim 算法测试样例 1

样例输入	设计理由	理论输出	样例输出

[(0,1,3),(0,2,9),(0,5,6),(1,2,9),(1,3,9),(1,4,2),(1,5,4),(2,3,8),(2,9,18),(3,4,8),(3,6,7),(3,8,9),(3,9,10),(4,5,2),(4,6,9),(5,6,9),(6,7,4),(6,8,5),(7,8,1),(7,9,4),(8,9,3)]	检查算法是否正确	Kruskal Algorithm: Prim Algorithm: The edges of the MST: [(0, 1), (1, 4), (4, 5), (3, 4), (3, 6), (6, 7), (7, 8), (8, 9), (2, 3)] Sum of the weight of MST: 38	Prim Algorithm: The edges of the MST: [(0, 1), (1, 4), (4, 5), (3, 4), (3, 6), (6, 7), (7, 8), (8, 9), (2, 3)] Sum of the weight of MST: 38 [Finished in 0.3s]

由于理论输出与样例输出相符，所以 Prim 算法测试样例 1 验证成功。

Prim 算法测试样例 2

样例输入	设计理由	理论输出	样例输出
（随机生成的有 500 个定点和 1000 条边的权重随机的图）	检查算法用时	计算所需的时间比 Kruskal 算法更长	Time: Prim: 2619 ms [Finished in 3.0s]

由于理论输出与期望相符，所以 Prim 算法测试样例 2 验证成功。

综上，Prim 算法通过所有样例的测试。

七、结果分析

1、Kruskal 算法正确性证明：

假设用 Kruskal 算法得到的T中有一边e不在真正的 MST 中，则将该边加入真正的 MST 中。加入之后 MST 中必然出现环。对出现的这个环，根据 Kruskal 算法，环中权重最大的那条边不会被加入（因为其他权重更小的边在未成环时被加入）。因为边e是由 Kruskal 算法加入T中的，故边e不可能是环中权重最大的那条边。则在有环的 MST 中删去权重最大的那条边，能得到一个更小的生成数，这与 MST 本身的矛盾。故 Kruskal 算法能够最小生成树。

2、Kruskal 算法的时间复杂度证明

算法根据边权对边进行归并排序的时间复杂度为$O(\log|E|)$。

算法在 STEP2～6 进行了|E|次循环，每次循环均调用了 Find 函数和 root_node 函数。Find 函数中只有比较和更改数据，复杂度为$O(1)$，root_node 函数由于递归调用自身查找根节点，故复杂度为$O(\log|V|)$。因此算法在 STEP2～6 的时间复杂度为$O(|\mathrm{E}|\log|V|)$。

因此 Kruskal 算法的复杂度为$O(|\mathrm{E}|\log|V|)$。

3、Kruskal 空间复杂度证明

算法所需的输入是边集，故空间复杂度为$O(|E|)$。

求节点根节点时使用列表存储每个节点的父节点，故空间复杂度为$O(|V|)$。

故算法的空间复杂度为$O(n)$。

4、Prim 算法正确性证明：

当点集大小为 2 时，由 Prim 算法可知找到的边e一定是两点之间最小的。

当点集大小为n时，假设 Prim 算法正确。

当点集大小为$n+1$时，可知 Prim 算法生成的前$n-1$条边满足最小。第n条边一定是从已经生成的 MST 中的一个节点和第$n+1$个节点组成的，由于 Prim 算法选取的是已选点集和未选点集组成的边中权重最小的一条，故此时第n条边一定是最短的一条。则生成的树也是最小的。

综上可知算法正确。

5、Prim 算法的时间复杂度证明：

Prim 算法中有$|V|$次循环，每一次循环中存在一次遍历，复杂度为$|E|$。故算法的时间复杂度为$O(|V||E|)$。

6、Prim 算法的空间复杂度证明：

算法用于记录图的边集、MST 的点集、边集需要的复杂度均为为$O(n)$，

八、总结

本次实验实现了 Kruskal 和 Prim 两个经典的求解 MST 问题的算法，并且 Kruskal 算法通过并查集的方法可以降低空间复杂度。而 Prim 算法一般来说由于每次都要遍历只有一个点在 X 中的边，故算法复杂度相比 Kruskal 算法来说较高，但是在交流讨论时，我也看到了其他同学的巧妙设计，利用不同的数据结构，例如优先队列等算法，能够很大降低时间复杂度，这些方法值得掌握。

选做题

一、实验题目：K 最小算法的实现

二、实验目的与内容

1、实验目的：

本次实验的目的是：了解 K-min 算法的内容，掌握 K-min 算法的具体实现方法，分析 K-min 算法的时间复杂度。

2、实验内容：

两人一组，各自实现 K-min 算法，分析复杂度、证明算法的正确性，共同完成 PowerPoint 进行展示交流。

三、算法设计

1、K-min 算法描述

K_min()程序描述：

STEP　1: 读入一个没有重复数字的数字串S和 k 的数值。

STEP　2: 调用 Select(S)函数挑选一个数字作为 pivot。

STEP　3: 利用 pivot 将数字串分成三组：小于 pivot 的数字串S_L，等于 pivot 的数字，以及等于 pivot 的数字串S_R。

STEP　4: 当$|S_L|$等于 k – 1 时，返回 pivot。

STEP　5: 当$|S_L|$大于 k – 1 时，返回 K_min(S_L, k)。

STEP　6：当$|S_L|$小于 k – 1 时，返回 K_min($S_R, k - |S_L| - 1$)。

Select()程序描述：

STEP　1：将字符串S分成五个数字一组，最后一组有$S \% 5$个数字。

STEP　2：对每一组的数字进行排序，挑出他们的中位数组成S'。

STEP　3：当$|S'| > 1$时，递归调用 Select(S')并返回

STEP　4：当$|S'| == 1$时，返回$S'[0]$。

2、算法流程图

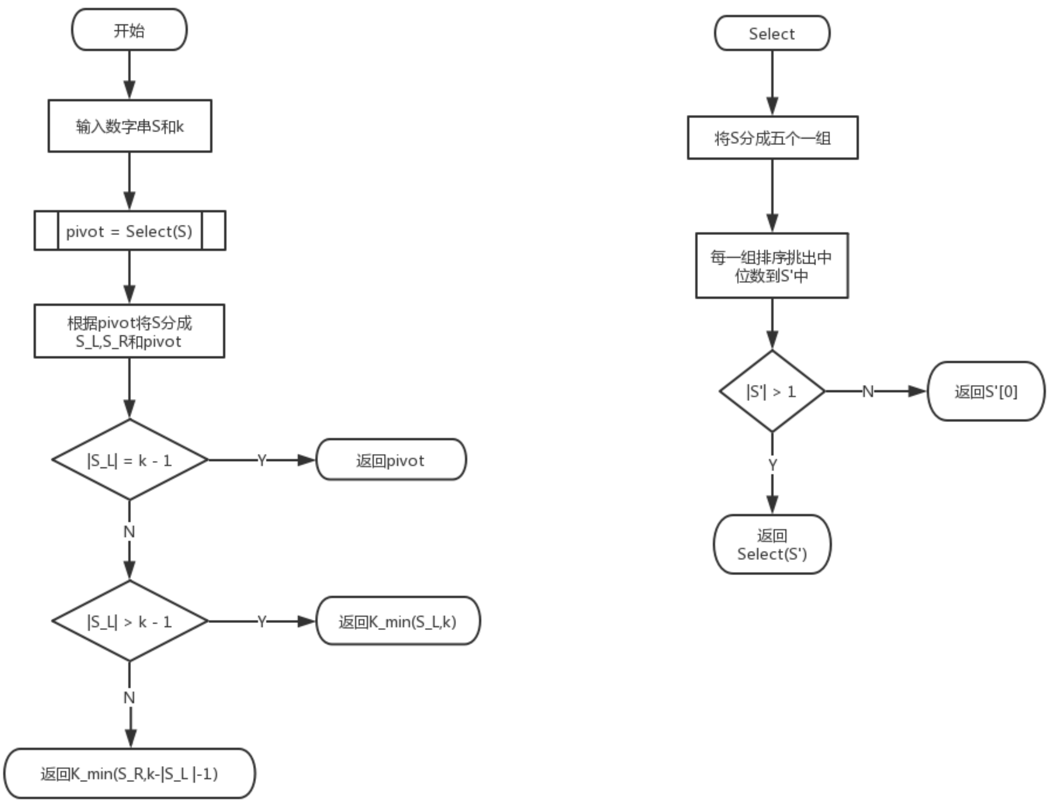

图 2.1 K-min 算法流程图

四、实验环境

操作系统：Windows 10
编译环境：Sublime Text3
语言版本：python3.7.4

五、实验过程

程序代码如下：

K-min 算法：

```
def K_Min(elements, Kth):
    smaller_than_pivot = []
    bigger_than_pivot = []
    equal_pivot = []
    piovt = select(elements, 5)
    if Kth < 1 or Kth > len(elements):
        return 'Kth out of range'
    for element in elements:
        if element == piovt:
            equal_pivot.append(element)
        elif element < piovt:
            smaller_than_pivot.append(element)
        else:
            bigger_than_pivot.append(element)
    if len(smaller_than_pivot) + 1 == Kth:
        return equal_pivot[0]
    elif len(smaller_than_pivot) + 1 < Kth:
        return K_Min(bigger_than_pivot, Kth - len(smaller_than_pivot) - 1)
    else:
        return K_Min(smaller_than_pivot, Kth)
```

Select 函数：

```
def select(elements, GroupNum):
    # level += 1
    Groups = []#用于将数列分组，每组最多 GroupNum 个
    MidNum_in_Groups = []
    i = 0
    while i * GroupNum < len(elements):
        #GroupNum 个一组，分成二维数组
        if i * GroupNum < len(elements) - 1:
            Groups.append(elements[i * GroupNum: i * GroupNum +
GroupNum])
        else:
```

```
            Groups.append(elements[i * GroupNum: len(elements)])
        i += 1
    # print('level', level,' group :\t', Groups)
    for group in Groups:
        #每一组排序找出中位数，加入中位数 group
        group.sort()
        MidNum_in_Groups.append(group[int(len(group) / 2)])
    # print('level', level, 'MidNum_in_Groups :\t',MidNum_in_Groups)
    if len(MidNum_in_Groups) != 1:
        #若中位数 group 的长度大于 GroupNum，递归寻找中位数
        return select(MidNum_in_Groups, GroupNum)
    else:
        #print('level', level, ' return num:\t',
MidNum_in_Groups[int(len(MidNum_in_Groups) / 2)])
        #否则直接找出中位数并返回
        return MidNum_in_Groups[0]
```

六、算法测试

测试样例 1

样例输入	测试理由	理论输出	样例输出
S = [5, 10, 1, 6, 14, 8, 3, 25, 32, 11,9, 17, 21, 30, 19, 7, 0, 38, 28, 16,31, 42, 37, 26, 15, 24, 12, 27, 4, 20,29]， K = 19	验证算法正确性	21	21 [Finished in 0.3s]

由于理论输出与样例输出相符，所以测试样例 1 验证成功。

测试样例 2

样例输入	测试理由	理论输出	样例输出
9 串随机生成的数字串	测试算法性能	数字串长度和计算用时组成的元组列表	[(915, 1), (1881, 1), (3346, 2), (3682, 2), (4205, 3), (4267, 3), (4638, 3), (5754, 4), (6036, 4)] [Finished in 1.8s]

由于理论输出与样例输出相符，所以测试样例 2 验证成功。

综上，算法通过所有样例的测试。

七、结果分析

1、算法正确性证明：

算法使用了分治策略，每一次通过 pivot 将 S 分组，可以确定 K-min 的位置。当$|S_L|$ > k – 1 时，则仅仅只需要对 S 中前$|S_L|$小的数字进行查找，当$|S_L|$ = k – 1，显然第 k 小的数字就是 pivot；当$|S_L|$ < k – 1 时，则仅仅只需要对 S 中前$|S_R|$大的数进行查找第$k-|S_L|-1$小的数。

2、算法时间复杂度分析：

Select 函数的复杂度为：$T(n)=T\left(\frac{n}{5}\right)+O\left(25\times\frac{n}{5}\right)=O(n)$

因此 K-min 算法的复杂度最差为：$T(n)=T\left(\frac{3n}{4}\right)+O(n)=O(n)$

3、算法空间复杂度分析：

算法以数组形式存放数字串，空间复杂度为$O(n)$，Select 函数的空间复杂度为$O=\frac{n}{5}+\frac{n}{25}+\cdots+\frac{n}{5^{\log_5 n}}=\frac{n}{4}-\frac{1}{4}=\ O(n)$。

故空间复杂度为$O(n)$。

八、总结

K-min 算法用到了分治策略，巧妙的将大问题缩小成小问题求解。其中在求解时间复杂度时，通过分治方法可以快速得出等式，并利用 Master Methord 进行快速求解，是相当快速的方法。