**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(是的权重。初始化MST的边集。

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，若E中没有未选择过的边则至STEP6。

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

STEP 5: 将边加入边集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：

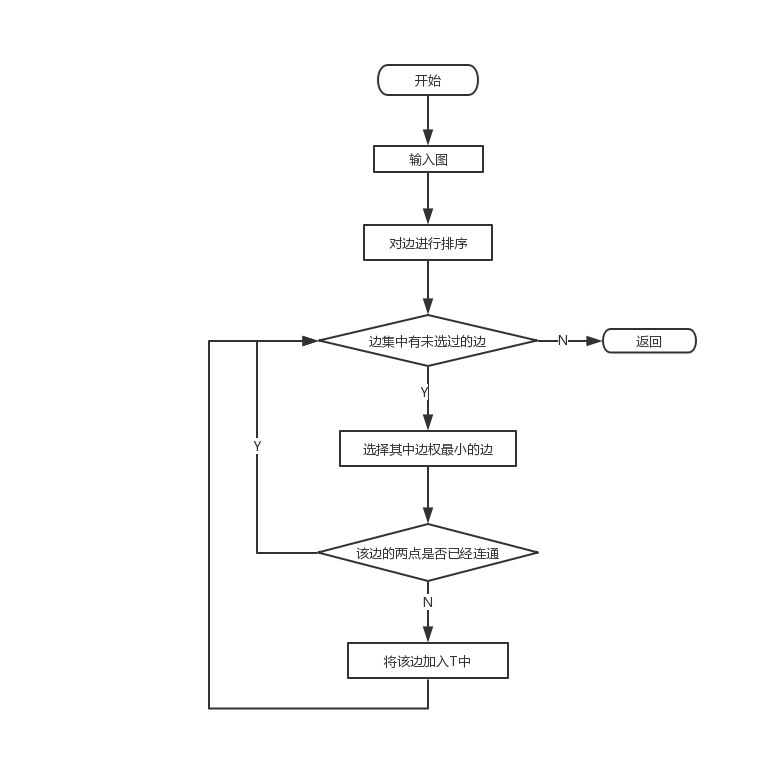


图1.1 Kruskal算法流程图

3、Prim算法描述

程序描述：

STEP 1: 以(是的权重。初始化MST的边集。为中任意一点。

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

STEP 3: 将边加入边集，将边中的不在中的点加入中。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：

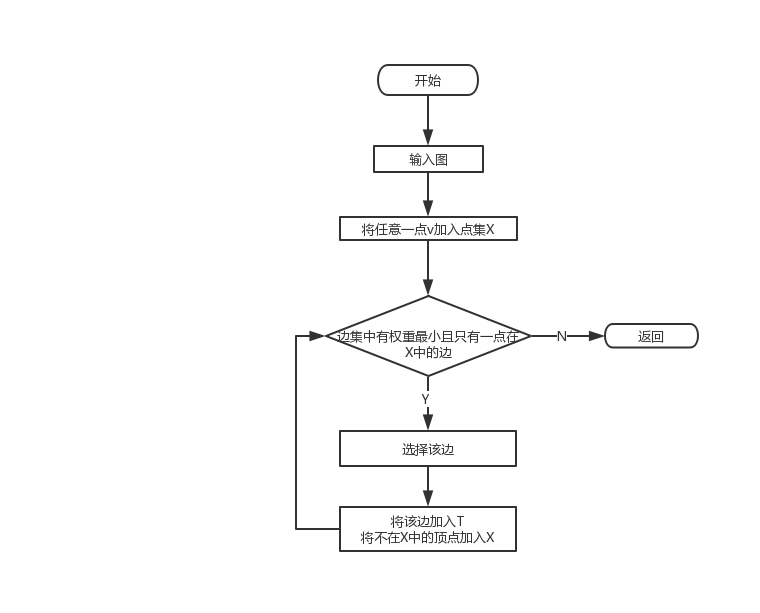


图1.2 Kruskal算法流程图

## 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 |  |

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 |  |

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 |  |

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 |  |

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算法得到的中有一边不在真正的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）。因为边是由Kruskal算法加入中的，故边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。

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，root\_node函数由于递归调用自身查找根节点，故复杂度为。因此算法在STEP2 ~ 6的时间复杂度为。

因此Kruskal算法的复杂度为。

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。

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。

故算法的空间复杂度为。

4、Prim算法正确性证明：

当点集大小为2时，由Prim算法可知找到的边一定是两点之间最小的。

当点集大小为时，假设Prim算法正确。

当点集大小为时，可知Prim算法生成的前条边满足最小。第条边一定是从已经生成的MST中的一个节点和第个节点组成的，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算法中有次循环，每一次循环中存在一次遍历，复杂度为。故算法的时间复杂度为。

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，

## 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和k的数值。

STEP 2: 调用Select()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，等于pivot的数字，以及等于pivot的数字串。

STEP 4: 当等于k – 1 时，返回pivot。

STEP 5: 当大于k – 1时，返回K\_min()。

STEP 6：当小于k – 1时，返回K\_min()。

Select()程序描述：

STEP 1：将字符串分成五个数字一组，最后一组有个数字。

STEP 2：对每一组的数字进行排序，挑出他们的中位数组成。

STEP 3：当时，递归调用Select()并返回

STEP 4：当时，返回。

2、算法流程图

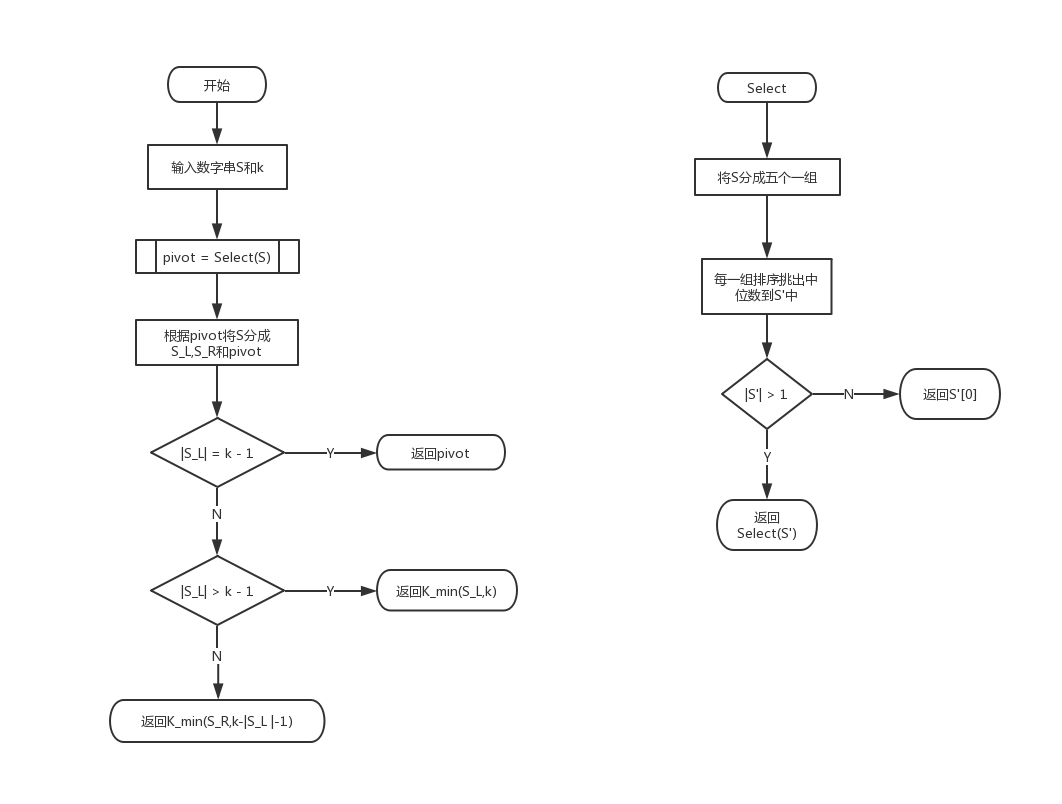


图2.1 K-min算法流程图

## 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 |  |

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 |  |

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 > k – 1 时，则仅仅只需要对S中前小的数字进行查找，当 = k – 1，显然第k小的数字就是pivot；当 < k – 1时，则仅仅只需要对S中前大的数进行查找第小的数。

2、算法时间复杂度分析：

Select函数的复杂度为：

因此K-min算法的复杂度最差为：

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，Select函数的空间复杂度为。

故空间复杂度为。

## 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。