
# 东北大学秦皇岛分校

# 计算机与通信工程学院

# 数据结构课程设计

设计题目 最小生成树问题

| | |
|---|---|
| 专业名称 | 计算机科学与技术 |
| 班级学号 | 170124 |
| 学生姓名 | **朱子权** |
| 指导教师 | 马海涛 |
| 设计时间 | **2019 年 1 月 2 日—2019 年 1 月 9 日** |

# 前言

数据结构课程主要是研究非数值计算的程序设计问题中所出现的计算机操作对象以及它们之间的关系和操作的学科。数据结构是介于数学、计算机软件和计算机硬件之间的一门计算机专业的核心课程，它是计算机程序设计、数据库、操作系统、编译原理及人工智能等的重要基础，广泛的应用于信息学、系统工程等各种领域。

学习数据结构是为了将实际问题中所涉及的对象在计算机中表示出来并对它们进行处理。通过课程设计可以提高学生的思维能力，促进学生的综合应用能力和专业素质的提高。通过此次课程设计主要达到以下目的：

了解并掌握数据结构与算法的设计方法，具备初步的独立分析和设计能力；

初步掌握软件开发过程的问题分析、系统设计、程序编码、测试等基本方法和技能；

提高综合运用所学的理论知识和方法独立分析和解决问题的能力；

训练用系统的观点和软件开发一般规范进行软件开发，培养软件工作者所应具备的科学的工作方法和作风。

训练学生灵活应用所学数据结构知识，独立完成问题分析，结合数据结构理论知识，编写程序求解指定问题。

# 课程设计任务书

**专业：**计算机科学与技术 **学号：**20178859 **学生姓名：** **朱子权**

## 设计题目：最小生成树问题

## 一、条件

本实验是基于装有 **Ubuntu(18.04 x64), Python(3.6.7), Graphviz(2.40.1-5), tkinter (3.6.7-1~18.04 amd64)** 的 PC 环境下进行的。

## 二、任务及要求

**任务：**

若要在 n 个城市之间建设通信网络，只需要假设 n-1 条线路即可。如何以最低的经济代价建设这个通信网，是一个网的最小生成树问题。

**要求：**

1. 利用普里姆、克鲁斯卡尔算法求网的最小生成树。
2. 实现教材 6.5 节中定义的抽象数据类型 MFSet。以此表示构造生成树过程中的连通分量。
3. 以文本的形式输出生成树中各条边以及他们的权值。（a , c , 3）
4. 利用堆排序实现选择权值最小的边。

## 三、设计主体

### 1. 需求分析

假如需要在 n 个城市之间建设通信网络，要做的是如何以最低的经济代价建设这个通信网，可以等效为一个网的最小生成树问题。根据任务要求，可以把任务分为以下四个子任务：（1） 利用普里姆、克鲁斯卡尔算法求网的最小

生成树。（2） 实现教材 6.5 节中定义的抽象数据类型 MFSet。以此表示构造生成树过程中的连通分量。(3) 以文本的形式输出生成树中各条边以及他们的权值。形如: (a , c , 3) (4) 利用堆排序实现选择权值最小的边。

根据程序需求，完成了程序主体，实现了所有的程序需求。除此之外。为了更直观的向用户展示，加入了图形界面和进行了对图和最小生成树的算法可视化。程序的功能图和流程图如下：

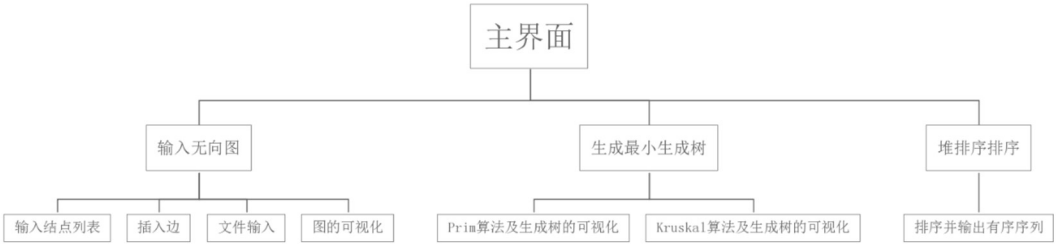


图 1 程序的功能图

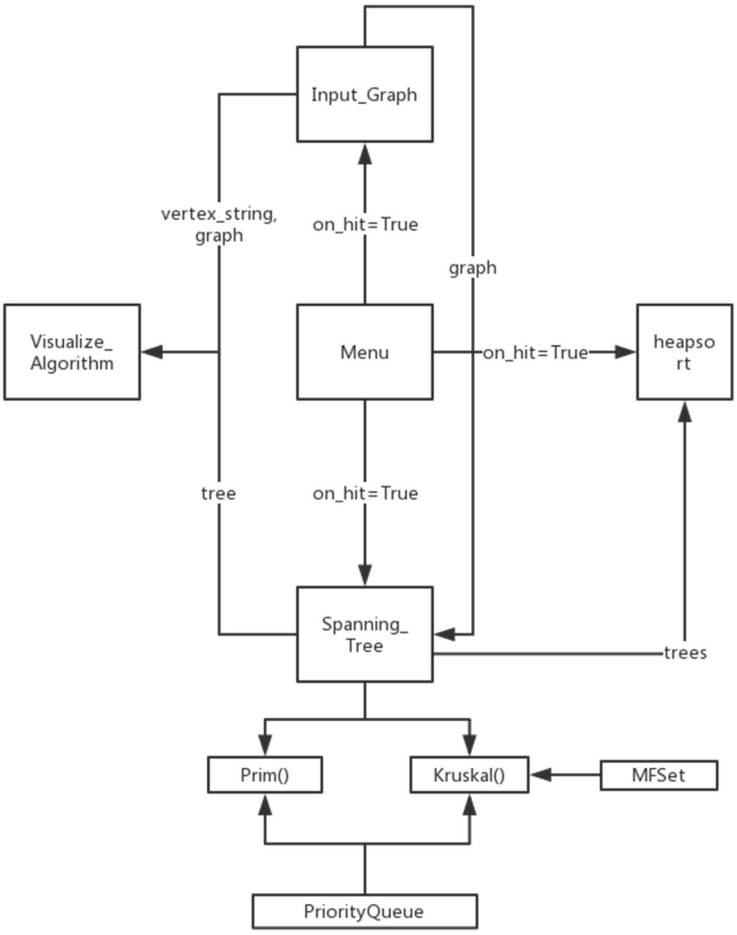


图 2 程序的流程图

为了方便使用和展示，程序提供了友好的图形界面，其中主菜单界面如下：


图 3 主界面

点击不同的按钮，我们可以实现不同的功能：其中“输入图”是输入我们所要使用的图，“可视化最小生成树”是选择 Prim 和 Kruskal 算法分别生存图的最小生成树，“对生成树进行堆排序”是根据前面生成的树通过堆排序，对边按照权值进行有序输出。

我们规定，在“输入图”的界面里，输入图的方式有从输入框输入和文件输入两种。其中“输入结点的字符串”输入框输入规定输入必须是一个字符串，每一个字符代表一个结点序号，“输入边的信息”输入框每次只能输入一条边，其中边必须满足格式如右：起点，终点 ，权重 的形式，其中起点终点必须是字符，权重必须是数字。在文件输入的方法中，文件的第一行是一个结点的字符串，接下来的每一行都是一条边，同样的每条边都满足 起点，终点 ，权重 的形式。“输入图”界面如下：

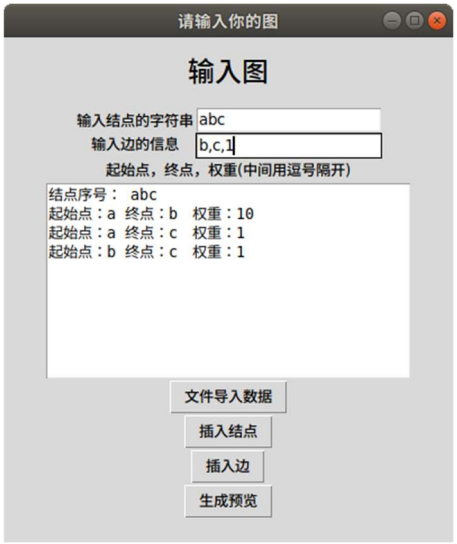


图 4 “输入图”界面

当选用文件输入时，因为文件输入会覆盖前面通过输入框所输入的所有的内容，

程序会产生警告：

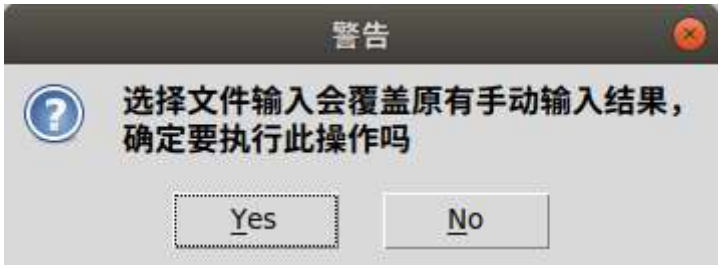


图 5　文件输入警告

点击生成预览按钮我们可以生成所输入图的可视化表示：

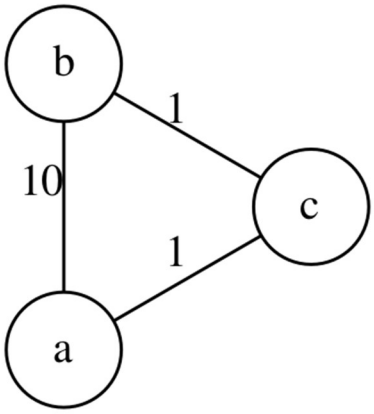


图 6　一个简单的可视化图的样例

点击“可视化最小生成树”，我们可以进入算法的选择界面，可以选择使用 Prim 或者 Kruskal 算法生成最小生成树。选择好算法之后，程序会输出生成的最小生成树，并可视化，用红色的边在图上代表最小生成树。

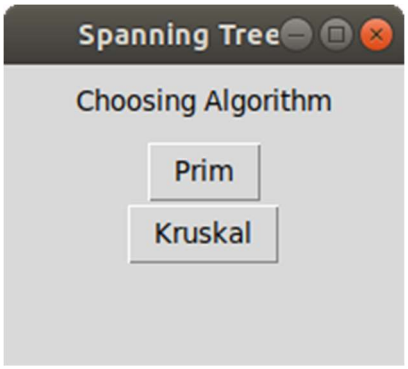


图 7 选择生成最小生成树算法

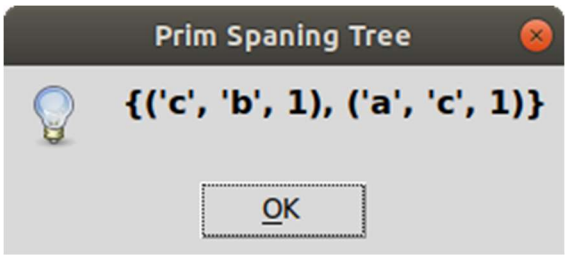


图 8　弹窗输出最小生成树连通分量

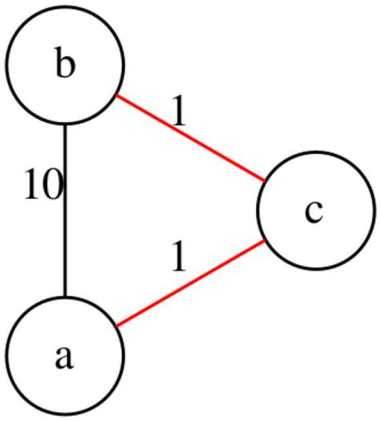


图 9 对于上面的图例，用红色标出其最小生成树的边

接下来我们点击“对生成树进行堆排序”我们可以生成其按照边权值增序排序的最小生成树。


图 10 输出已经排序好的生成树连通分量

## 2. 系统设计

本系统的函数之间的调用关系图如下：

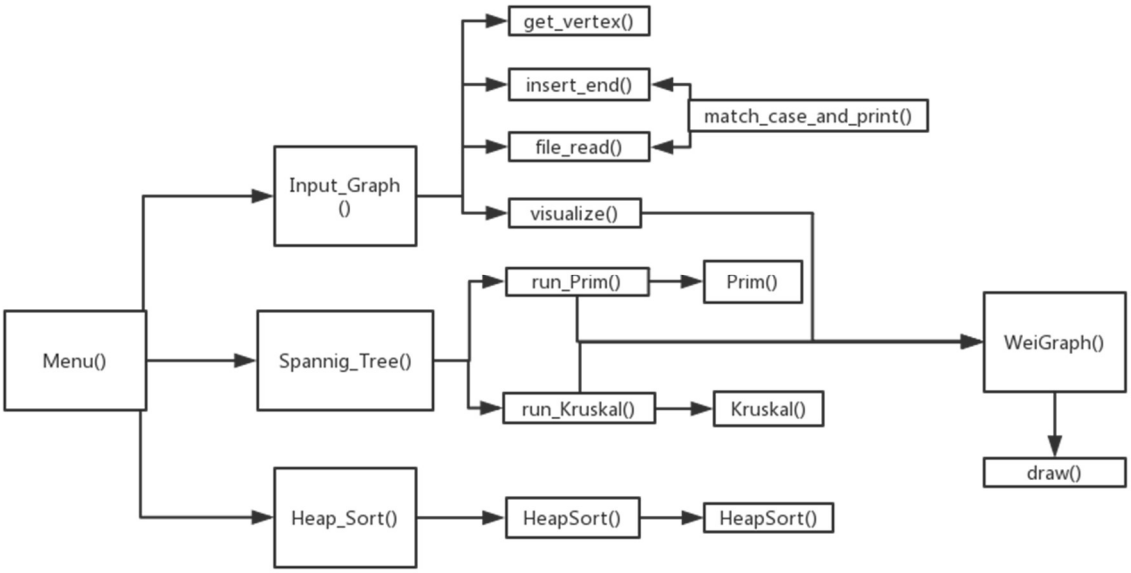


图 11 函数之间的调用关系图

程序所使用的数据结构如下：

**并查集：**在本程序中，并查集用于表示在 Kruskal 算法中生成的最小生成树。并查集处理的是集合之间的关系。在这种数据类型中，N 个不同元素被分成若干个组，每组是一个集合，这种集合叫做分离集合。并查集支持查找一个元素所属的集合和两个元素分别所属的集合的合并。
并查集支持以下操作：
Initial(X):建立一个仅有成员 X 的新集合。
Merge(X,Y)：将包含 X 和 Y 的动态集合合并为一个新集合 S，此后该二元素处于同一集合。
Find(X)：返回一个包含 X 的集合。

数据结构定义：

```
ADT MFSet{：
    Initial();
    Merge();
    Find();
}
```

基本操作：Initial() 初始化并查集 sets

```
def Initial(sets, item):
    Sets.add(item)
```

Merge() 合并两个并查集

```
def Merge():
        a = Find(item_a)
        b = Find(item_b)
        sets.remove(a)
        sets.remove(b)
        sets.add(a+b)
```

Find() 查找元素时候在哪个并查集中，如果是返回其所在的并查集，否则返回 None

```
def Find():
        for s in self.sets:
            if s.issuperset(item):
                return s
        return None
```

**优先队列：**普通的队列是一种先进先出的数据结构，元素在队列尾追加，而从队列头删除。在优先队列中，元素被赋予优先级。当访问元素时，具有最高优先级的元素最先删除。优先队列具有最高级先出的行为特征。我们在 Kruskal 和 Prim 算法中都有优先队列的使用。

数据结构定义：

```
ADT PriorityQueue{
put();
get();
top();
}
```

基本操作： put() 把元素放入优先队列中
get() 得到优先队列的第一个元素出队
top() 得到第一个元素，但不出队

```
def top():
    size = queue.size()
    top_node = get()
    put(top_node)
    for i in range(size-1):
        put(get())
    return top_node
```

**3. 系统实现**

**核心算法：**

**Prim 算法：**

1. 算法描述

1).输入：一个加权连通图，记作 G

2).初始化：将除起点以外的访问标志 visit 全部设置为 0

3).重复下列操作，直到所有结点全部进入 all_nodes：

a. 将当前结点的所有未访问的邻接结点全部加入优先队列中

b. 清除所有优先队列中前面剩下的终点被访问的边

c. all_nodes 加入权值最小边，即 PriorityQueue 中第一个出队列的边

4).all_nodes 就是我们要求的最小生成树

2. 利用优先队列的 Prim 算法的时间复杂度是 O(n^2), 空间复杂度是 O(n)

3. 伪代码实现

```
def Prim(G):
    all_nodes = set()
    length = len(G.nodes)
    total = 1
    visit = {}
    s = G.nodes[0]
    queue = PriorityQueue()
    for node in G.nodes:
        visit[node] = 0
    visit[s] = 1
    while total < length:
        for (node, w) in G.adjs(s):
            if visit[node] == 0:
                nn = (s, node, w)
                queue.put(nn)
        while not queue.empty() and visit[queue.top()[1]]:
            queue.get()
    nn = queue.get()
    all_nodes.add(nn)
    s = nn[1]
    visit[s] = 1
    total += 1
    return all_nodes
```

**Kruskal 算法：**

1. 算法描述

1).输入：一个加权连通图，记作 G

2).重复下列操作，直到所有结点全部进入 all_nodes：

a. 每次取得权值最小的边 ，即优先队列的队首

b. 利用并查集判断加入的边的两个端点是否来自同一连通分量（实质就是保证加入边后不会产生回路）

3).all_nodes 就是我们要求的最小生成树

2. 利用并查集的 Kruskal 算法的时间复杂度是 O(|E|log|E|)，其中 E 分别是图的边集

3. 伪代码实现

```
def Kruskal(G):
    all_nodes = MFSet()
    for node in G.nodes:
        all_nodes.Initial(node)
    queue = PriorityQueue()
    for edge in G.edges:
        queue.put(edge)
    while not queue.empty():
        (u, v, w) = queue.get()
        if all_nodes.Find(u) != all_nodes.Find(v):
            all_nodes.edge_set.add((u, v, w))
            all_nodes.Merge(u, v)
    return all_nodes
```

**堆排序算法：**

1. 算法描述

1).输入：一个加权连通图，记作 G

2).初始化：根据边的权值，调整前 n/2 的元素，使其成为大顶堆

3).重复 n-1 次，将堆顶元素和堆尾元素互换，重新调整除被换下来堆顶以外的所有元素，使其仍然是个大顶堆

4).最后生成的是一个按权值递增排序的序列

2. 利用堆排序算法的时间复杂度是 O(nlog(n)), 空间复杂度是 O(1)

3. 伪代码实现

调整堆：

```
def HeapAdjust(trees, low, high):
    t_low = low
```

```
            child = 2 * t_low # left
            temp = trees[low]
            while child <= high:
                if child<high and trees[child][2]<self.trees[child + 1][2]:
                    child += 1 # right
                if temp[2] < self.trees[child][2]:
                    self.trees[t_low] = self.trees[child]
                    t_low = child
                    child *= 2
                else:
                    break
            self.trees[t_low] = temp
```

进行堆排序:

```
    def HeapSort(trees):
            result = []
            for i in range(trees.size//2, 0, -1):
                HeapAdjust(i, trees.size)
            for i in range(trees.size, 1, -1):
                trees[1], trees[i] = trees[i], trees[1]
            HeapAdjust(1, i-1)
            for edge in trees:
                if edge:
                    result.append(edge)
            return result
```

## 4. 用户手册

### (1) 运行程序

我们在目录下打开终端， 输入下面的命令：

Python3 Menu.py

接着会打开程序的主界面

### (2) 输入数据

点击“输入图”按钮，进入输入图的界面。输入所有结点后，点击“插入结点”。接着在输入边的信息，每输入一条边，点击“插入边”。这样就完成了对图的插入操作。

同时对于有大量数据的用户来说，我们可以通过文件输入来导入图。点击“文件导入数据”，选择文件，即完成操作。

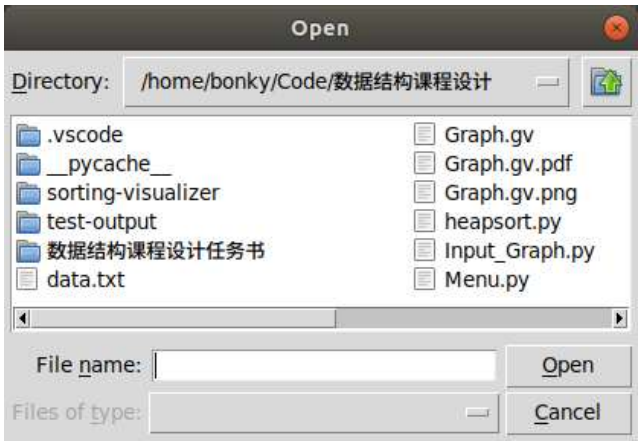


图 12 文件选取界面

最后别忘了点击“生成预览”确认文件是否生成成功。

### (3) 生成最小生成树

返回主界面，点击“可视化最小生成树”，选择你所需要的算法，会返回最小生成树所有的边，和在图上的可视化表示。

### (4) 对最小生成树进行堆排序

返回主界面，点击“对生成树进行堆排序，会返回堆排序的结果。

## 5. 测试

利用附录 1 中的 Data 数据对程序进行测试：

### （1）输入过程

为了节省输入时间，我们利用文件输入：

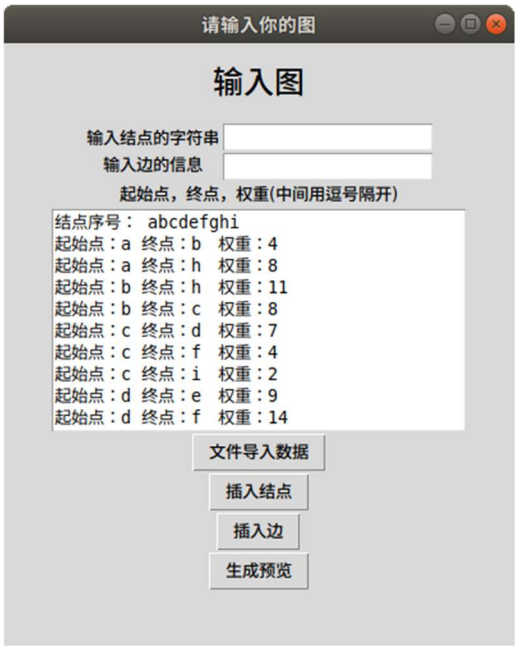


图 13 文本框显示插入的结点

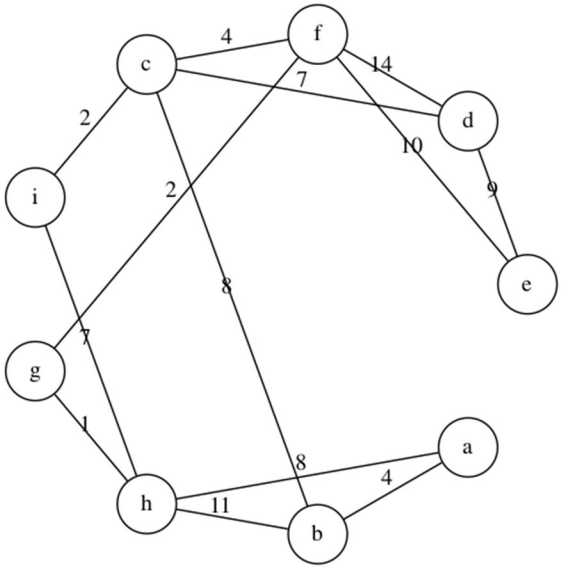


图 14 输入的图预览如图

## (2）生成最小生成树

生成的最小生成树如下：

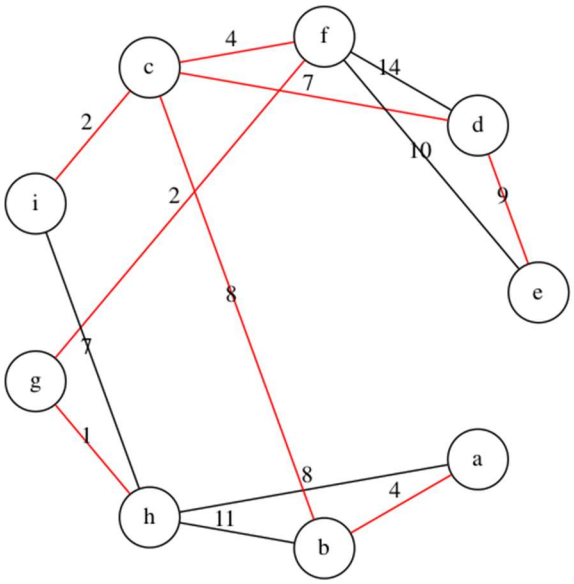


图 15　最小生成树

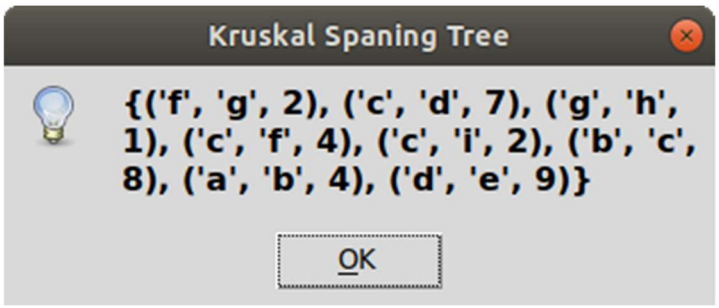


图 16 最小生成树(利用 Kruskal)

### (3）堆排序排序

堆排序序列如下：

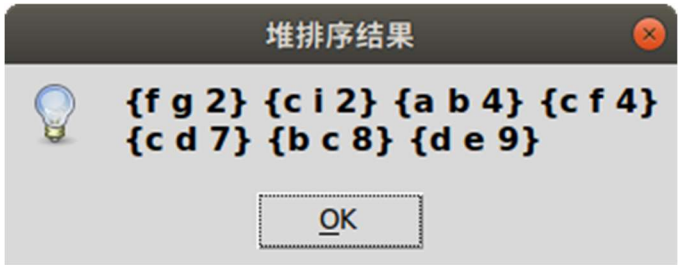


图 16 堆排序结果

## 四、结束语

本次的数据结构课程设计针对具体的实际项目来进行需求分析，测试计划，概

要设计，详细设计，测试分析等具体的步骤流程走下来，从中我收获了很多

经过一周的课程设计，有很多的心得体会。首先，在编写函数之前要充分利用图书资源和网络资源；其次，应该更详细的考虑实际情况，才能使程序更切合实际，更具有实用性；更多的我想应该是组员之间的合作精神吧！

通过这次课程设计练习，使我更深刻地理解了数据结构重要存储结构的的精髓-----线性链表的使用。完成整个程序设计有，对线性链表的使用掌握的更加熟练。

同时通过直接对模拟最小生成树的生成和堆排序，加深了对数据结构的理解和认识。并在完成课程设计的过程作主动查阅了相关资料，学到了不少课本上没有的技术知识。

经过这次课程设计，我深刻认识到算法在程序设计中的重要性，一个完整的程序总是由若干个函数构成的，这些相应的函数体现了算法的基本思想。

通过此次课程设计，我了解了编写应用软件的一般步骤，获得了很多宝贵的经验。特别是怎样将理论与实践相结合，把书本上的的内容应用到我们做的程序上。怎样使各个子模块实施其详细功能，特别是各个子模块之间的接口，一定要相当清晰达到相互协调的作用。其次我熟悉了数据结构的知识学会了很多关于程序设计的经验和技巧，明白了程序设计的使用性和通用性事程序生存周期长短的关键，学会了程序调试的一般方法。重要的是通过本次程序设计，我逐步具备了走向程序员的基本素质。知道如何在困难重重时一步一步细心发现问题，解决问题。知道了在软件设计中对界面和功能如何平衡，如何达到相对完美。

## 五、参考资料

1. 数据结构（C 语言版）/严蔚敏，吴伟民编著。----清华大学出版社，2007
   http://www.tup.com.cn
2. 数据结构题集/史嘉权 编著。 -------清华大学出版社，2008

## 六、附录

**Data**

abcdefghi
a,b,4
a,h,8
b,h,11
b,c,8
c,d,7
c,f,4
c,i,2
d,e,9
d,f,14
e,f,10
f,g,2
g,h,1
h,i,7

**Menu.py**

```
import tkinter as tk
import Visualize_Algorithm as va
from Input_Graph import Input_Graph
from Spanning_Tree import Spanning_Tree
from heapsort import Heap_Sort

class Menu:

    def __init__(self):
        '''
        生成图形界面
        - 输入图
        - 生成最小生成树
        - 输出堆排序结果
        '''

        window = tk.Tk()
        window.title("朱子权-20178859-数据结构课设")
        window.geometry("400x300")
        l = tk.Label(window,
                     text='最小生成树模拟程序',
                     font=('WenQuanYi MicroHei', 18),       # 字体和字体大小
                     width=40, height=2   # 标签长宽
                     )
        l.pack()
        Graph = Input_Graph(window)
        Tree = Spanning_Tree(window, Graph)
        Heap_Sort(window, Tree.tree)
        window.mainloop()


Menu()
```

**Input_Graph.py**

```
import tkinter as tk
import Visualize_Algorithm as va
import re
import tkinter.messagebox as message
import tkinter.filedialog as tkfile

class Input_Graph:
    '''
```

```
    输入图
    '''

    def __init__(self, window):
        '''
        vertex_string 输入结点 （字符串表示）
        graph 边的集合
        match 正则表达式匹配的 pattern
        '''

        self.window = window
        self.on_hit = False
        self.vertex_string = ''
        self.graph = set()
        tk.Button(window,
                  text='输入图',
                  width=15, height=2,
                  command=self.hit).pack()
        self.match = re.compile("([a-zA-Z]+),([a-zA-Z]+),([0-9]+)")

    def hit(self):
        def match_case_and_print(string):
            '''
            对于输入的形式（‘str’, 'str', ‘str’），进行（‘str’, 'str', int）形式转化存入 graph
            '''

            matched = re.findall(self.match, string)

            if matched:
                a = (matched[0][0], matched[0][1], int(matched[0][2]))
                self.graph.add(a)
                t.insert('end', '起始点：' + a[0])
                t.insert('end', '\t 终点：' + a[1])
                t.insert('end', '\t 权重：'+str(a[2])+'\n')

            else:
                message.showwarning('输入不合法', '按照起始点，终点，权重， 中间用逗号隔开格式输入')

        def get_vertex():
            '''
            得到结点列表， 并输出
            '''
```

```
        self.vertex_string = er.get()

        if self.vertex_string:
            t.insert('end', '结点序号：\t'+self.vertex_string+'\n')

    def insert_end():
        '''
        得到边， 并输出
        '''

        var = e.get()
        match_case_and_print(var)

    def visualize():
        '''
        可视化所输入的图
        '''

        wg = va.WeiGraph(self.vertex_string, self.graph)
        wg.draw().view()

    def file_read():
        '''
        利用文件输入的方式输入图
        '''

        file_name = tkfile.askopenfilename()

        judge = message.askyesno('警告', '选择文件输入会覆盖原有手动输入结果，确定
要执行此操作吗')

        if judge:
            file = open(file_name).read().split('\n')
            self.graph = set()
            self.vertex_string = file[0]
            if self.vertex_string:
                t.insert('end', '结点序号：\t'+self.vertex_string+'\n')

            for i in range(1, len(file)):
                match_case_and_print(file[i])

    if self.on_hit == False:
        entry_window = tk.Tk()
        entry_window.title("请输入你的图")
```

```
entry_window.geometry("400x450")

tk.Label(entry_window,
         text='输入图',
         font=('WenQuanYi MicroHei', 18),
         width=40, height=2
         ).pack()

frm = tk.Frame(entry_window)
frm.pack()
frm_l = tk.Frame(frm)
frm_r = tk.Frame(frm)
frm_l.pack(side='left')
frm_r.pack(side='right')

tk.Label(frm_l, text='输入结点的字符串').pack()
er = tk.Entry(frm_r, show=None)
er.pack()

tk.Label(frm_l, text='输入边的信息').pack()
e = tk.Entry(frm_r, show=None)
e.pack()

tk.Label(entry_window, text='起始点，终点，权重(中间用逗号隔开)').pack()

t = tk.Text(entry_window, height=10, width=40)
t.pack()

tk.Button(entry_window,
          text="文件导入数据",
          command=file_read
          ).pack()

tk.Button(entry_window,
          text="插入结点",
          command=get_vertex
          ).pack()

tk.Button(entry_window,
          text="插入边",
          command=insert_end
          ).pack()

tk.Button(entry_window,
```

```
                text="生成预览",
                command=visualize
                ).pack()

        entry_window.mainloop()
```

**Visualize_Algorithm.py**

```
from graphviz import Digraph, Graph


class WeiGraph():
    '''
    带权图
    '''

    def __init__(self, nodes, edges):
        '''
        nodes  结点集
        edges  边集
        graph  邻接表
        G       Graphviz.Graph  用于可视化图
        '''

        self.nodes = nodes
        self.edges = edges
        self.graph = []

        for node in nodes:
            self.graph.append([(node, 0)])

        for (x, y, w) in edges:
            for n in self.graph:
                if n[0][0] == x:
                    n.append((y, w))
                if n[0][0] == y:
                    n.append((x, w))

        self.G = None

    def adjs(self, node):
        '''
        返回邻接表，  用于 Prim 算法
```

```
        param node ： 需要寻找邻接点的结点

        return ： 邻接结点表
        '''

        return list(filter(lambda n: n[0][0] == node, self.graph))[0][1:] #返回以 node 开始的邻接结点表

    def draw(self, color_filter=None):
        '''
        可视化所生成的图
        '''

        if color_filter is None:
            color_filter = lambda edge: 'black' #全部为黑色

        settings = dict(name='Graph', engine='circo', node_attr=dict(shape='circle'))
        self.G = Graph(**settings)

        for node in self.nodes:
            self.G.node(str(node), str(node))
        for (x, y, w) in self.edges:
            self.G.edge(str(x), str(y), label=str(w), color=color_filter((x, y, w)))

        return self.G


def cf(tree):
    '''
    最小生成树用红色标出
    '''

    def color_filter(edge):
        in_tree = filter(lambda t: t[:2] == edge[:2] or t[:2][::-1] == edge[:2],tree)

        if len(list(in_tree)) > 0:
            return 'red'
        return 'black'

    return color_filter
```

**Spaning_Tree.py**

```
import tkinter as tk
import MFSet
```

```
import Visualize_Algorithm as va
import tkinter.messagebox as message


class Spanning_Tree:
    '''
    最小生成树
    '''

    def __init__(self, window, Graph):
        '''
        Gragh  需要生成最小生成树的图
        tree  存储所生成的最小生成树
        '''

        self.window = window
        self.Graph = Graph
        self.on_hit = False
        self.tree = set()
        tk.Button(window,
                  text='可视化最小生成树',
                  width=15, height=2,
                  command=self.hit).pack()

    def hit(self):

        def run_Prim():
            '''
            执行 Prim 算法，  并执行最小生成树的可视化
            '''

            wg = va.WeiGraph(self.Graph.vertex_string, self.Graph.graph)
            self.tree = set(MFSet.Prim(wg))

            wg.draw(color_filter=va.cf(self.tree)).view()
            message._show('Prim Spaning Tree', self.tree.__str__())

            with open("tree.txt", 'w') as file:
                file.write(str(self.tree))

        def run_Kruskal():
            '''
            执行 Kruskal 算法，  并执行最小生成树的可视化
            '''
```

```
                wg = va.WeiGraph(self.Graph.vertex_string, self.Graph.graph)
                self.tree = set(MFSet.Kruskal(wg).edge_set)

                wg.draw(color_filter=va.cf(self.tree)).view()
                message._show('Kruskal Spaning Tree', self.tree.__str__())

                with open("tree.txt", 'w') as file:
                    file.write(str(self.tree))

            if self.on_hit == False:
                entry_window = tk.Tk()
                entry_window.title("Spanning Tree")
                tk.Label(entry_window,
                         text='Choosing Algorithm',
                         font=('WenQuanYi MicroHei', 10),
                         width=40, height=2
                         ).pack()
                entry_window.geometry("200x150")

                tk.Button(entry_window,
                          text="Prim",
                          command=run_Prim
                          ).pack()

                tk.Button(entry_window,
                          text="Kruskal",
                          command=run_Kruskal
                          ).pack()

                entry_window.mainloop()
```

**MFset.py**

```
from queue import PriorityQueue as PQ


class PriorityQueue(PQ):
    '''
    优先队列， 继承 Python 标准类 PriorityQueue， 将按照权值小的排序
    '''

    def __init__(self, *option):
        super().__init__(*option)
```

```
    def put(self, item):
        '''
        将结点按照权值在前的放入优先队列

        param item：  要放入队列的边
        '''

        super().put(item[::-1])

    def get(self):
        '''
        删除并得到队列的第一个元素

        return ：  按照原样输出边 （起点, 终点, 权值） （tuple）
        '''

        return super().get()[::-1]

    def top(self):
        '''
        得到第一个元素但不在队列删除

        return top_node：  输出队列第一个元素 （起点, 终点, 权值） （tuple）
        '''

        size = self.qsize()
        top_node = self.get()
        self.put(top_node)

        for i in range(size-1):
            self.put(self.get())

        return top_node


class MFSet:
    '''
    并查集， 用于存储在 Kruskal 算法产生的树
    '''

    def __init__(self):
        '''
        sets  并查集
        edge_set  边集
```

```
        '''

        self.sets = set()
        self.edge_set = set()

    def Initial(self, item):
        '''
        初始化并查集， 将 item 加入并查集

        param item： 加入并查集的元素 (tuple)
        '''

        self.sets.add(frozenset(item))

    def Merge(self, item_a, item_b):
        '''
        合并 a 和 b 两个并查集

        param item_a, item_b : 需要被合并的两个并查集 （set）
        '''

        a = self.Find(item_a)
        b = self.Find(item_b)

        self.sets.remove(a)
        self.sets.remove(b)

        self.sets.add(a.union(b))

    def Find(self, item):
        '''
        判断 item 在哪个并查集中

        param item ： 查询的元素 （set）

        return s ： 返回 item 所包括的集合
        '''

        for s in self.sets:
            if s.issuperset(frozenset(item)):
                return s

        return None
```

```
    def __str__(self):
        string = ''

        for edge in self.edge_set:
            string += str(edge)
            string += '\n'

        return string


def Kruskal(G):
    '''
    利用优先队列和并查集， 执行 Kruskal 算法

    param G : 带权图 (WeiGraph)

    return all_nodes: 返回用并查集表示的最小生成树 (MFSet)
    '''

    all_nodes = MFSet()

    for node in G.nodes:
        all_nodes.Initial(node)

    queue = PriorityQueue()

    for edge in G.edges:
        queue.put(edge)

    while not queue.empty():
        (u, v, w) = queue.get()
        if all_nodes.Find(u) != all_nodes.Find(v):
            all_nodes.edge_set.add((u, v, w))
            all_nodes.Merge(u, v)

    return all_nodes


def Prim(G):
    '''
    利用优先队列， 执行 Prim 算法

    param G : 带权图 (WeiGraph)
```

```
    return all_nodes: 返回用集合表示的最小生成树 (set)
    '''

    all_nodes = set()
    length = len(G.nodes)
    total = 1
    visit = {}
    s = G.nodes[0]

    queue = PriorityQueue()

    for node in G.nodes:
        visit[node] = 0
    visit[s] = 1

    while total < length:
        for (node, w) in G.adjs(s):
            if visit[node] == 0:
                nn = (s, node, w)
                queue.put(nn)

        while not queue.empty() and visit[queue.top()[1]]: # 防止边访问不到的情况
            queue.get()

        nn = queue.get()
        all_nodes.add(nn)
        s = nn[1]
        visit[s] = 1
        total += 1

    return all_nodes
```

**heapsort.py**

```
import tkinter as tk
import tkinter.messagebox as message
import re


class Heap_Sort:
    '''
    执行堆排序
    '''

    def __init__(self, window, tree):
```

```
        '''
        t_trees 临时存储 trees
        trees 存储最小生成的树， 第 0 个元素是()，保证下标从 1 开始
        tree——size 树的大小
        '''

        self.window = window
        self.match = re.compile("\'([a-zA-Z]+)\', \'([a-zA-Z]+)\', ([0-9]+)")
        self.trees = [()]
        self.tree_size = 0
        self.on_hit = False
        tk.Button(self.window,
                  text='对生成树进行堆排序',
                  width=15, height=2,
                  command=self.hit).pack()


    def hit(self):
        '''
        按下按钮操作， 弹出窗口展示结果
        '''
        self.trees = [()]
        t_trees = [()]+re.findall(self.match, open('tree.txt').read())
        self.tree_size = len(t_trees)-1
        for i in range(1, self.tree_size+1):
            self.trees.append(
                (t_trees[i][0], t_trees[i][1], int(t_trees[i][2])))

        result = self.HeapSort()
        message._show('堆排序结果', result)

    def HeapAdjust(self, low, high):
        '''
        调整堆，使其为大顶堆

        param : 调整的范围（low～high）
        '''

        t_low = low
        child = 2 * t_low    # left
        temp = self.trees[low]

        while child <= high:
            if child < high and self.trees[child][2] < self.trees[child + 1][2]:
```

```
                child += 1 # right

            if(temp[2] < self.trees[child][2]):
                self.trees[t_low] = self.trees[child]
                t_low = child
                child *= 2
            else:
                break

        self.trees[t_low] = temp

    def HeapSort(self):
        '''
        进行对权值的堆排序

        return result :  堆排序的结果  (list)
        '''

        result = []

        for i in range(self.tree_size//2, 0, -1):
            self.HeapAdjust(i, self.tree_size)

        for i in range(self.tree_size, 1, -1):
            self.trees[1], self.trees[i] = self.trees[i], self.trees[1]
            self.HeapAdjust(1, i-1)

        for edge in self.trees:
            if edge:
                result.append(edge)

        return result
```