**课程：算法分析与设计**

**班级：计算141 学号： 1404030137 姓名：周朋飞**

**实验日期：2016 年 11 月 6 日 交报告日期： 11.6**

|  |
| --- |
| **实验(No. 4 )题目：学习和掌握贪心算法** |
| **实验目的及要求：**  **目的：**  **1、深入分析和理解问题，了解和学习贪心算法；**  **2、设计算法，给出算法的伪代码；**  **3、给出源代码和执行测试用例；**  **4、分析算法的复杂度，给出复杂度的阶。** |
| **实验内容及步骤：**  1. 构建huffman树  **def** creat\_tree(dict):  tree = []  **while** len(dict) > 1:  list = sorted(dict.iteritems(), key=**lambda** d: d[1], reverse=False)  a = list[0]  b = list[1]  dict.pop(a[0])  dict.pop(b[0])  **if** len(a[0]) == 1:  tree.append(a[0])  **if** len(b[0]) == 1:  tree.append(b[0])   **if** a[1] > b[1]:  tmp = a[0] + b[0]  **else**:  tmp = b[0] + a[0]  total = a[1] + b[1]   tree.append(tmp)  dict.update({tmp: total})  **print** tmp,**":"**,total  **return** tree  这个函数按照二叉树先序遍历的方式，将二叉树存入一个线性表中。  由于每次都使用一次排序选出最小值，该函数的时间复杂度为O(N^2) ~ O(N!)  程序截图    上述函数每次从字典中取出key最小的值，所以用优先队列代替每一次的排序  **def** creat\_tree\_v2(list):  tree = []  q = queue.PriorityQueue()  **for** i **in** list:  q.put(i)  *# print i* **while** q.qsize() > 1:  min1 = q.get()  min2 = q.get()   **if** len(min2[1]) == 1:  tree.append(min1[1])  tree.append(min2[1])  **else**:  tree.append(min2[1])  tree.append(min1[1])  total = min1[0] + min2[0]  code = min2[1] + min1[1]  i = (total,code)  q.put( i )  *# print (total,code)* **print** total,**" "**,min2[1],**" "**,min1[1]  tree.append(q.get()[1])  **return** tree  优先队列每一次插入的时间复杂度为O(logn)，所以总体的时间复杂度为O(nlogn)  函数运行截图    2.将huffman树翻译为huffman code  **def** find\_huffcode(list):  huff\_code = { list[0]: **''**}  *# huff\_code.update({list[0]:''})* **for** postion **in** range(1,len(list)):  pre = postion - 1  **while**( list[postion] **not in** list[pre] ):  pre -= 1  **if** pre < 0:  **break   if** pre > -1:  **if** pre == postion - 1:  huff\_code.update({ list[postion]: huff\_code[list[pre]] + **'0'** } )  **else**:  huff\_code.update({list[postion]: huff\_code[list[pre]] + **'1'**})   **print "pos = "**,postion, **" pre = "**,pre, **"code = "**,huff\_code[list[postion]]   **return** huff\_code  如果线性表中 后一位置的字符串是前一位置的子串，则后一位置的huffman\_code 为前一位+“1”  若不是子串，则寻找其父节点，直至找到包含其字符串的节点。其huffman\_code为该节点+”0”  函数运行截图：      若以每一次 “in语句” 为基本运算，则该程序的时间复杂度为O(nlogn)  3.计算压缩率  **def** test\_huffcode(list, huffcode):  length\_pre = 0;  length\_now = 0;  **for** i **in** list:  length\_pre += 8 \* i[0]  length\_now += len( huffcode[i[1]] ) \* i[0]   **return** length\_pre, length\_now  以每一个字符占8位为例      若将每一次in运算的时间复杂度记为O(N)，则整个程序的时间复杂度为O(n^2logn) |
|  |

注：《实验内容及步骤》项目的内容如果较多，可以加附页。

|  |
| --- |
| **实验结果及心得:**  **在构建huffman树的时候，虽然脑子里想象的非常具体，树状、左节点大于右节点。但是在实现的时候，写得代码却有很大的变化。**  **分析问题时的步骤可以化简，只要程序的每一个步骤没有二义性，能达到预期的目标，能解决该问题，这段程序就是可行的。** |
| **教师评语：**  **成绩:**  **教师签字： 年 月 日** |