第9章 贪婪技术

*Steven*

## 1. 以金额和硬币面额作为输入，得到最少找零硬币数。为以上找零问题设计一个贪婪算法，给出算法伪码，并分析其时间复杂度。

**（1）代码**

|  |
| --- |
| **def** changes\_arrangement(n, changes: list = **None**):  *"""  使用贪心思想，求解最小张数的找零方案，及对应张数  当然，众所周知，贪心思想不一定能求得最优解* **:param** *n: 找零金额* **:param** *changes: 零钱面额，升序排列，且认为所有零钱数量都是无限的* **:return***: 贪心算法计算的最小找零张数，及对应的找零方案  """* changes = [1, 5, 10, 20, 50, 100] **if** changes **is None else** changes  *# 默认的零钱面额情况* changes.sort() *# 升序排列* res = [] *# 找零方案* i = len(changes) - 1 *# 一个计数器，表示从面值最大的零钱开始考虑* **while** n != 0: *# 停止条件* **if** n >= changes[i]: *# 应找的零钱比当前大* n = n - changes[i]  res.append(changes[i])  **else**: *# 当前零钱大于应找的面额，则应重新考虑找零面额* i = i - 1  **return** len(res), res |

**（2） 测试结果**

|  |  |  |  |
| --- | --- | --- | --- |
| 输入 | 7 | 13 | 43 |
| 输出 |  |  |  |

**（3） 时间复杂度分析**

该算法的基础操作是while循环中的判断语句。由于通常来说零钱中最小都为1，所以最坏情况为判断语句执行了n次，即找零方案全为1,故其时间复杂度为。

## 2. 如果在单处理器上，有个运行时间分别为的已知作业，请考虑安排一个调度计划，使得所有作业花费在系统中的时间最少（一个作业花费在系统中的时间是该作业用于等待的时间和用于运行的时间之和）。为该问题设计一个贪婪算法，讨论所设计的贪婪算法是否总能产生最优解。

该问题由于是在“单处理器”上，即认为只有单线程，所以安排方式应将各个任务线性排列。

对于任意一种安排方式，其任务编号从到，则易知第个任务的等待时间为其前面的个任务的运行时间之和，故第个任务花费在系统中的时间为，全部任务花费在系统中的时间为。

故若要使总时间最短，应使按照运行时间**升序排列**任务的执行计划。

由于上述讨论是在任意条件下进行的，所以上述结论对任意情况均有效，即是说该贪婪算法总能产生最优解。

## 3. 使用Kruskal算法求解下图的最小生成树，给出求解过程。

图示

描述已自动生成

**（1）代码**

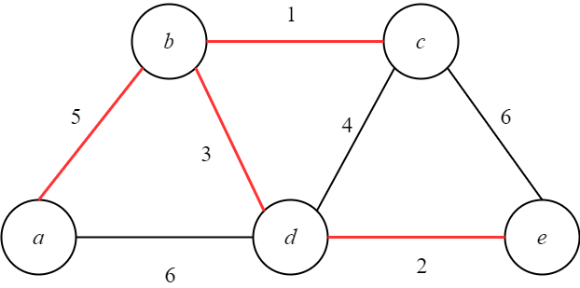
|  |
| --- |
| **def** Kruskal(graph):  *"""  使用Kruskal算法求解最小生成树* **:param** *graph: 邻接矩阵，不存在边的边权用I(inf)代替* **:return***: 最小生成树（所有边）  """  # 最小生成树的边集，每个边都使用[边权, 边起点, 边终点]的三元组的表示形式* tree = []  *# 原图的点集、边集* Vex, Edge = set([i **for** i **in** range(len(graph))]), []  *# 边集初始化* **for** row **in** range(len(graph)):  **for** col **in** range(row + 1, len(graph)):  **if** graph[row][col] != I:  Edge.append([graph[row][col], row, col])  Edge.sort(key=**lambda** i: i[0]) *# 按照升序排列  # 将最小边放入最小生成树* tree.append(Edge.pop(0))  Vex = Vex - {tree[-1][1], tree[-1][2]} *# 更新Vex，即删去取出的边对应的两点* **while** len(tree) != len(graph) - 1:  **for** i **in** Edge:  **if** (i[1] **not in** Vex **and** i[2] **in** Vex) **or** (i[1] **in** Vex **and** i[2] **not in** Vex): *# 判断是否成环* tree.append(i)  Vex.remove(i[1] **if** i[1] **in** Vex **else** i[2])  Edge.remove(i)  **break  return** tree |

**（2）测试结果**

图片包含 文本

描述已自动生成

即表示最小生成树的边为**(*b, c*)**、**(*b, d*)、(*d, e*)、(*a, b*)**。



## 4. 设计一个求加权连通图的最大生成树算法，其中，最大生成树是包含最大可能权重的树。

将上述Kruskal算法中排序换为降序排列即可。

**（1）代码**

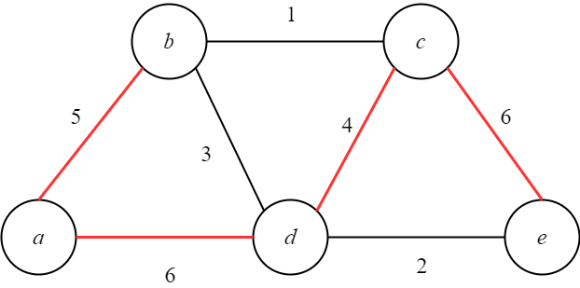
|  |
| --- |
| **def** Kruskal\_reverse(graph):  *"""  使用Kruskal算法求解最大生成树* **:param** *graph: 邻接矩阵，不存在边的边权用I(inf)代替* **:return***: 最大生成树（所有边）  """  # 最大生成树的边集，每个边都使用[边权, 边起点, 边终点]的三元组的表示形式* tree = []  *# 原图的点集、边集* Vex, Edge = set([i **for** i **in** range(len(graph))]), []  *# 边集初始化* **for** row **in** range(len(graph)):  **for** col **in** range(row + 1, len(graph)):  **if** graph[row][col] != I:  Edge.append([graph[row][col], row, col])  Edge.sort(key=**lambda** i: i[0], reverse=**True**) *# 按照降序排列  # 将最大边放入最大生成树* tree.append(Edge.pop(0))  Vex = Vex - {tree[-1][1], tree[-1][2]} *# 更新Vex，即删去取出的边对应的两点* **while** len(tree) != len(graph) - 1:  **for** i **in** Edge:  **if** (i[1] **not in** Vex **and** i[2] **in** Vex) **or** (i[1] **in** Vex **and** i[2] **not in** Vex): *# 判断是否成环* tree.append(i)  Vex.remove(i[1] **if** i[1] **in** Vex **else** i[2])  Edge.remove(i)  **break  return** tree |

**（2）测试结果**

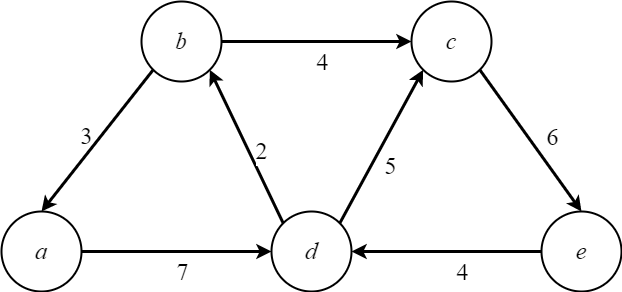
图片包含 图表

描述已自动生成

即表示最小生成树的边为**(*a, d*)**、**(*a, b*)、(*c, d*)、(*c, e*)**。



## 5. 求解以下单源最短路径问题的实例，以顶点a作为起点，给出求解过程。

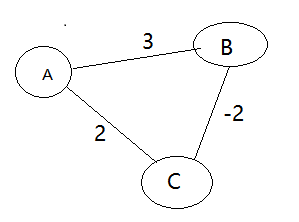


**求解步骤：**

|  |
| --- |
| 以为起点的不重复最短路径为，长度为7  拓展结点，当前最短路径为，长度为7 |
| 以为起点的不重复最短路径为，长度为2  拓展结点，当前最短路径为，长度为9 |
| 以为起点的不重复最短路径为，长度为4  拓展结点，当前最短路径为，长度为13 |
| 以为起点的不重复最短路径为，长度为6  拓展结点，当前最短路径为，长度为19 |

## 6．给出一个反例，说明对于包含负权重的加权连通图，Dijkstra算法可能会无效。

在如图所示的简单无向图中，若是起点为A，使用Dijkstra算法的拓展顺序为A-C-B，可得A-C最短距离为2，A-B最短距离为0。但实际上A-C最短距离为A-B-C中的1，故证得Dijkstra算法不能用于带负权重的带权无向图。



7.

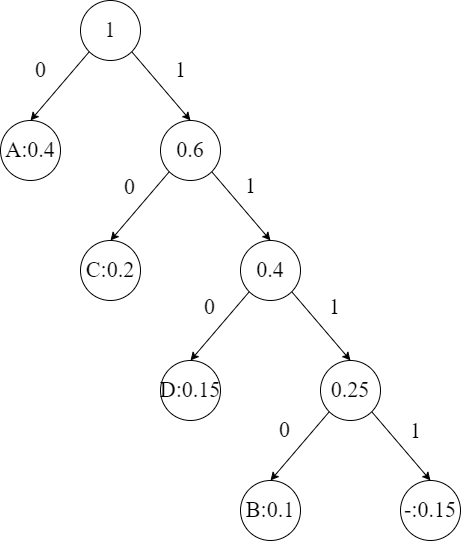
**（1）对于下面的数据构造哈夫曼编码：**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **字符** | **A** | **B** | **C** | **D** | **-** |
| **出现概率** | **0.4** | **0.1** | **0.2** | **0.15** | **0.15** |

**（2）用（1）中的编码对文本ABACABAD进行编码；**

**（3）对于编码为100010111001010文本用（1）中的编码进行解码。**

**（1）**

****

**（2）**

根据上面的计算，各字母的哈夫曼编码结果如下：

|  |  |
| --- | --- |
| A | 0 |
| B | 1110 |
| C | 10 |
| D | 110 |
| - | 1111 |

则ABACABAD的编码结果为：0|1110|0|10|0|1110|0|110

**（3）**

100010111001010划分为10|0|0|10|1110|0|10|10=CAACBACC