**算法设计与分析 实验报告**

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 姓名 | 王少博 | | | 院系 | | 软件学院 | | | 学号 | | 181110315 |
| 任课教师 | | 王金宝 | | | | | 指导教师 | | 王金宝 | | |
| 实验地点 | | 研究院1号楼517 | | | | | 实验时间 | | 2020年 10 月 31 日 | | |
| 实验名称 | | 树搜索算法设计与实现 | | | | | | | | | |
| 同 组 人 | | 无 | | | | | | | | | |
| 实验内容（问题，思路，程序，结果） | | | | | | | | | | | |
| **问题**   * **活动选择问题**   1. 输入：*S={1, 2, …, n}*, *F*={ [*si，fi*] }，*n≥i≥1*   输出：*S*中的最大相容活动集合  要求:  （1）使用动态规划技术求解活动选择问题；（优化解的结构和递归方程可以参考课件，建议阅读算法导论16.1节，其中包含一种更有效的方式 （更加建议自行分析优化解的结构并设计优化解的递归方程）. 请写出你使用的递归方程，说明其中各项的含义; 请说明优化子结构以及问题是否具有子问题重叠性; 实现你设计的算法.）  （2）使用贪心算法求解活动选择问题；（实现课件上的算法即可）  （3）使用树搜索算法求解活动选择问题；（说明如何将问题转化为树搜索问题；说明数中每个节点的含义；说明你使用的分支界限搜索算法）  （随机生成若干活动，并使用它测试动态规划算法、贪心算法和树搜索算法）  本次作业使用jupyter notebook编写。   1. **动态规划：**   **问题定义**：设表示活动i到活动j之间可以进行的活动集合（即满足的活动k的集合）。为了简便，我们定义。即加入活动0和活动n+1。  **递归方程**：我们设*dp[i][j]*表示活动i到活动j之间可以加入的活动个数，即最大的个数。定义如下：  **重叠子问题：**求解的时候需要用到其二维矩阵中左边和下面的所有元素，所以存在重叠子问题  **优化子结构：**若活动*k*为的优化解中的某个活动，则一定是的优化解，且一定是的优化解。  证明：如果不是的优化解，那么假设的优化解是，那么存在那么我们有，与是优化解矛盾！同理可以证明是优化解。  **代码：**  n,s,f = getDataset(mode='dp')    dp = np.zeros((n+2,n+2),dtype=int)  res = np.zeros((n+2,n+2),dtype=int)    #得到优化解 def get\_result(s,f,i,j,res,S):   if res[i][j]==0:   return S   key = res[i][j]   K = S.union({key})   return get\_result(s,f,i,key,res,K).union(get\_result(s,f,key,j,res,K))  #得到活动安排策略 def activity\_selection\_dp(s,f,n):   for k in range(2,n+2):   for i in range(n-k+2):   j = i + k   pos = -1   for t in range(i+1,j):   if s[t]>=f[i] and f[t]<=s[j]:   if dp[i][t]+dp[t][j]+1>dp[i][j]:   dp[i][j]=dp[i][t]+dp[t][j]+1   pos = t   if(pos!=-1):   res[i][j]=pos  # print(res)   S = {0,n+1}   S = get\_result(s,f,0,n+1,res,S)   return sorted(list(S-{0,n+1}))  **可视化：**     1. **贪心算法**   **贪心策略**：  将活动集合*S*按结束时间升序排列，每次选择不冲突的活动集合中结束时间最小的活动。  **剩余子问题：**  选择了活动*i*以后的剩余子问题为，易证*S*的优化解包括活动1。  **贪心选择性：**  设*A*是*S*的一个优化解且包括活动1，则是的优化解。  （证明在课件中给出，在此不再赘述）  代码：  n,s,f = getDataset(mode='greedy')    A = {0}  k=0  for m in range(1,n):   if s[m]>=f[k]:   A = A.union({m})   k = m  result\_greedy = list(A)  **可视化：**     1. **树搜索：**   问题转化：  选择活动i  不选择活动i  我们从第1个活动开始，作为根节点建立一个二叉搜索树。左边表示不选择这个活动，右边表示选择这个活动。由于已经建立了排序规则，所以如果我们不选择活动i，我们将活动i+1作为其左子树的根。否则查看下一个可以选择的活动作为选择的活动进行搜索。每次子树的高度为父亲树的高度-1，因此我们得到了如下计算最后树的高度（即最大选择的活动数）的算法如下：  n,s,f = getDataset(mode='tree')  #得到活动k之后可以选择的活动序列 def getPossibleNodes(s,f,i,n):   return [k for k in range(i+1,n) if s[k]>=f[i]]    visit = np.zeros(n)  def naive\_activity\_selection\_tree(s,f,k,n,h):   if k>=n:   return h   #不选   h\_left = naive\_activity\_selection\_tree(s,f,k+1,n,h)     #选择   nodes = getPossibleNodes(s,f,k,n)   if len(nodes)>0:   h\_right = naive\_activity\_selection\_tree(s,f,nodes[0],n,h+1)   else:   h\_right = h+1     if h\_left<h\_right:   visit[k]=1   return max(h\_left,h\_right)  **可视化**    但是这里我们对于每一种情况都进行了搜索，代价太大。我们采取一种分支界限的手段来减少代价。方法是，我们首先从活动1开始，随机找一个解（每次在可以选择的活动中随机选一个作为下一个选择的节点，按照顺序找到最后一个）得到了这个树的高度/深度记作h。这样我们在求解其他情况的时候，可以和这个h做一个比较，我们在这个节点之前已经选择的节点个数为a，剩余还可以选择的节点为b，那么我们知道如果a+b>=h 一定就不需要继续向下探索了，这个情况一定不是优化解。根据这个性质我们改进了一下算法如下：  visit = np.zeros(n)  #得到一组随机的解 def getNaiveSolution(s,f,k,n):   nodes = getPossibleNodes(s,f,k,n)   if len(nodes)==0:   return 1   idx = np.random.randint(len(nodes))   return getNaiveSolution(s,f,nodes[idx],n)+1    h\_bound = getNaiveSolution(s,f,0,n)    def activity\_selection\_tree(s,f,k,n,h):   if k>=n:   return h   #剪枝   h\_bound = getNaiveSolution(s,f,k,n)   if np.where(visit==1)[0].shape[0]+n-k<=h\_bound:   return h     #不选   h\_left = activity\_selection\_tree(s,f,k+1,n,h)     #选择   nodes = getPossibleNodes(s,f,k,n)   if len(nodes)>0:   h\_right = activity\_selection\_tree(s,f,nodes[0],n,h+1)   else:   h\_right = h+1     if h\_left<h\_right:   visit[k]=1   return max(h\_left,h\_right)  **可视化** | | | | | | | | | | | |
| 实验结论（结果分析、遇到的困难和解决方法等） | | | | | | | | | 备注 | |  |
| **结果分析：**  动态规划算法由于使用的n+2维度的数组，所以可视化结果的时候略有不同。同时树搜索算法由于是递归计算的，且可能存在多组解，所以结果和贪心算法可视化时可能出现不同（贪心算法其实就是树搜索中的其中一条分支）。  从复杂度而言，贪心算法最好，是O(n)级别的。而DP算法是O(n3)，树搜索算法是O(2n)的，复杂度很大。  **遇到的困难和解决方案：**  遇到的困难主要在于DP算法里面关于数组的定义，一开始希望用时间来定义，但是由于区间的长度不确定，很多时间点定义毫无意义。最后还是参考了算法导论使用了活动i和活动j之间的定义。更新数组的时候采用了对角线求解的方法（因为我们需要求解出二维数组左边和下面的数才能求解出这个数），同时我们还需要人为增添两个活动0，活动n+1。给编程时候的下标造成了一点困难。解决方案就是多谢了几次，考虑了边界条件。  树搜索的时候的考虑在于是否要建立树，实际上我们只需要保留选择的点就可以了，求树的深度也不需要实实在在地构建树，所以我没有通过手动建立树这个数据结构来求解。  经过此次实验，对于三个算法地理解更加透彻了。 | | | | | | | | | | | |
| 教师评价 | | | 备注 | |  | | | 得 分 | |  | |
|  | | | | | | | | | | | |