**集美大学实验报告**

**班级 计算2114 学号 202121331104 姓名 庄佳强 成绩**

**课程 《算法设计与分析》 日期 2024 年 4 月 30 日**

|  |  |  |  |
| --- | --- | --- | --- |
| **实验项目** | **实验四 回溯法的应用** | **指导老师** | **杨艳华** |
| **实验目的** | 掌握回溯法的使用 | | |
| **一、实验内容**  1.0/1背包问题。给定一载重量为W的背包及n个重量为wi、价值为vi的物体，1≤i≤n,要求重量和恰好为W具有最大的价值。  2.在国际象棋中，皇后是最厉害的棋子，可以横走、直走，还可以斜走。棋手马克斯·贝瑟尔 1848 年提出著名的八皇后问题：即在 8 × 8 的棋盘上摆放八个皇后，使其不能互相攻击 —— 即任意两个皇后都不能处于同一行、同一列或同一条斜线上  3.有n个作业（编号为1～n）要在由两台机器M1和M2组成的流水线上完成加工。每个作业加工的顺序都是先在M1上加工，然后在M2上加工。M1和M2加工作业i所需的时间分别为ai和bi（1≤i≤n）。 | | | |
| **二、问题分析**  1.  **问题描述：**  给定 N 个物品，每个物品有一定的重量和价值，目标是从中选择若干物品放入容量为 W 的背包中，使得背包内物品的总价值最大。  **解向量：**  一个长度为 N 的二进制数组，每个元素表示对应物品是否被选择放入背包中，通常用0表示不放入，1表示放入。  **解空间组织结构:**  解空间可以用树形结构来表示，树的每一层对应一个物品，每个物品有两种状态：放入或者不放入，因此这个树是一个二叉树。  **剪枝/限界函数：**  剪枝条件：如果当前背包的重量超过了背包的容量，可以直接剪枝，不再继续搜索。  **限界函数：**  如果剩余物品的总价值加上当前背包的总价值小于当前已知最优解，可以直接剪枝。  如果剩余物品的总重量加上当前背包的总重量小于背包总容量，可以直接剪枝。  2.  **问题描述：**  八皇后问题是经典的回溯算法应用示例，可以通过定义解向量、解空间组织结构和剪枝/限界函数来求解。在这个问题中，主要目标是将八个皇后放置在8×8的棋盘上，使得每个皇后都不能互相攻击，即任意两个皇后不能位于同一行、同一列或同一对角线上。  **解向量**  解向量通常是一个长度为8的一维数组，其中每个元素表示一个皇后在棋盘上的列位置，即解向量的第 i 个元素表示第 i 行皇后所在的列位置。这种表示方法确保了每个皇后都在不同的列上，并且不在同一条斜线上。  **解空间组织结构**  八皇后问题的解空间可以看作是一个树形结构：  树的层级：每一层代表棋盘上的一行，从第一行到第八行。  每个节点的分支：每个节点代表当前已经放置的皇后的位置情况。每一行的节点有8个可能的分支，即每一列可以放置皇后。  叶子节点：叶子节点表示一种完整的解，即每一行都放置了一个皇后。  **剪枝/限界函数**  在每一步放置皇后时，通过剪枝策略来减少搜索空间：  同一列：检查当前放置的皇后与之前所有皇后是否在同一列，如果是则剪枝。  同一斜线：检查当前放置的皇后与之前所有皇后是否在同一斜线上，如果是则剪枝。  3.  **问题描述：**  流水线调度问题可以通过回溯法来求解，通过定义解向量、解空间组织结构和剪枝/限界函数，可以有效地减少搜索空间，提高求解效率。  **解向量**  解向量是一个包含作业编号的排列序列，表示每个作业在流水线上的执行顺序。假设有 n 个作业，则解向量可以表示为一个长度为 n 的排列，其中第 i 个位置的值表示第 i 个作业在流水线上的执行顺序。  **解空间组织结构**  解空间是所有可能的作业排列组合的集合，可以描述为一个树形结构：  根节点：初始状态，即未对任何作业进行排序的状态。  分支节点：对当前解进行扩展得到的新解。每个节点可以通过将某个作业插入到当前作业序列的不同位置来生成分支节点。  叶子节点：无法再进一步扩展的节点，即已经达到了一个完整的作业排列序列，是一个潜在的解。  **剪枝/限界函数**  最小完成时间剪枝：如果当前正在构建的调度方案的完成时间已经超过当前最优解的完成时间，则可以剪枝，因为无论如何调整作业顺序，这个分支都不可能产生更好的解。 | | | |
| **三、数据结构定义**  1.  输入物品列表 arr，每个物品包含 (weight, value, selected)  m 表示当前处理的物品下标，初始值为 0  n 表示物品总数  W 表示背包容量  2.  flag = False  sum = 0  n = 8 # 假设 n 为 8  a = [0] \* (n + 1) # 存储每行皇后所在的列  b = [0] \* (n + 1) # 标记列是否被占用  c = [0] \* (2 \* n) # 标记正对角线是否被占用  d = [0] \* (2 \* n) # 标记反对角线是否被占用  3.  bestf = INF # 存放最优的调度时间  f1 = 0 # 当前调度在M1上的总执行时间  f2 = [0] \* MAX # 存放每层次的最优调度时间  x = [0] \* MAX # 当前的调度方案  bestx = [0] \* MAX # 存放当前作业的最优调度方案 | | | |
| **四、算法伪代码描述**  1.  # 主函数：背包问题求解  Function KnapSack(arr, m, n, W):  i = 0  j = 0  sum\_v = 0  sum\_w = 0    # 递归终止条件  If m == n:  For i = 0 To m - 1:  sum\_v += arr[i].value \* arr[i].selected  sum\_w += arr[i].weight \* arr[i].selected    If sum\_v > MaxValue And sum\_w == W:  MaxValue = sum\_v  For i = 0 To n - 1:  choose[i] = arr[i].selected  Else:  For j = 0 To 1:  arr[m].selected = j  If isOverweight(arr, m, W, n):  KnapSack(arr, m + 1, n, W)  # 判断是否满足约束条件  Function isOverweight(arr, m, W, n):  sum\_w = 0  sum\_v = 0    For i = 0 To m:  sum\_w += arr[i].selected \* arr[i].weight  sum\_v += arr[i].selected \* arr[i].value    For i = m + 1 To n - 1:  sum\_v += arr[i].value    If sum\_v < MaxValue:  Return False    If sum\_w > W:  Return False    Return True  2.  # 全局变量  flag = False  sum = 0  n = 8 # 假设 n 为 8  a = [0] \* (n + 1) # 存储每行皇后所在的列  b = [0] \* (n + 1) # 标记列是否被占用  c = [0] \* (2 \* n) # 标记正对角线是否被占用  d = [0] \* (2 \* n) # 标记反对角线是否被占用  # 打印当前解  Function print():  If flag:  Print newline  flag = True  For i = 1 To n:  For j = 1 To n:  If j == 1:  If a[i] == j:  Print "Q"  Else:  Print "."  Else:  If a[i] == j:  Print " Q"  Else:  Print " ."  Print newline  # 深度优先搜索放置皇后  Function dfs(i):  If i == n + 1:  sum = sum + 1  print()  Return  Else:  For j = 1 To n:  If b[j] == 0 And c[i + j] == 0 And d[i - j + n] == 0:  a[i] = j  b[j] = 1  c[i + j] = 1  d[i - j + n] = 1  dfs(i + 1)  b[j] = 0  c[i + j] = 0  d[i - j + n] = 0  # 初始化并开始DFS  Function solveNQueens():  flag = False  sum = 0  dfs(1)  Print "Total solutions:", sum  3.  # 交换x, y  Function swap(x, y):  temp = x  x = y  y = temp  # 深度优先搜索从第i层开始  Function dfs(i, n, a, b):  If i > n: # 达到叶子节点，产生一种调度方案  If f2[n] < bestf:  bestf = f2[n]  For j = 1 To n:  bestx[j] = x[j]  Else:  For j = i To n:  swap(x[i], x[j])  f1 += a[x[i]] # 在第i层选择执行作业x[i]，在M1上的执行时间  f2[i] = max(f1, f2[i - 1]) + b[x[i]]  If f2[i] < bestf:  dfs(i + 1, n, a, b) # 仅扩展调度时间小于bestf的节点  f1 -= a[x[i]] # 回溯，撤销对作业x[i]的选择  swap(x[i], x[j])  # 主函数  Function main():  n = Input() # 读取作业数  a = Array(n + 1) # 存放作业在M1上的执行时间  b = Array(n + 1) # 存放作业在M2上的执行时间  For i = 1 To n:  a[i], b[i] = Input() # 读取每个作业在M1和M2上的执行时间  f1 = 0  bestf = INF # 初始时的作业最优调度时间为无穷大  f2 = [0] \* (n + 1) # 初始f2数组中的执行时间为0  For k = 1 To n: # 初始时作业的执行顺序：1，2，...，n  x[k] = k  dfs(1, n, a, b) # 从作业1开始调度  Print(bestf)  For j = 1 To n:  Print(bestx[j], end=" ")  Print() | | | |
| **五、算法时间和空间复杂度分析（要求写出详细过程）**  1.  时间复杂度分析：  假设有 n 个商品，则 KnapSack 函数的时间复杂度为 O(2^n）。整体时间复杂度为O(2^n)。  空间复杂度分析：  使用了递归，递归栈的深度为n,整体空间复杂度为O(n)。  时间复杂度分析：  在没有任何剪枝的情况下，每行有 n 种选择，总共有 n! 种可能的排列组合。因此，最坏情况下的时间复杂度为 O(n!)。  空间复杂度分析：  最坏情况下，递归的深度为 n，即需要O(n) 的空间来存储递归调用栈的信息。总空间复杂度为 O(n)。  3.  时间复杂度分析：  流水线调度问题的时间复杂度主要取决于解空间的大小。在最坏情况下，所有 n! 个排列都需要被生成和评估。因此，最坏情况下的时间复杂度为 O(n!)。  空间复杂度分析：  递归调用栈的深度：在最坏情况下，递归的深度为 n，即需要O(n) 的空间来存储递归调用栈的信息。空间复杂度：总空间复杂度为 O(n)。 | | | |
| **六、错误记录和解决**  1.因为这是跟着书本上照着写的，没有什么问题。 | | | |
| **七、心得体会**  在实现流水线调度问题的过程中，我学到了许多关于深度优先搜索和剪枝策略的知识，并且对问题的解决方法有了更深入的理解。以下是我在实现过程中的一些心得体会：  剪枝策略的重要性：在搜索过程中，使用合适的剪枝策略可以极大地减少搜索空间，从而提高算法的效率。我通过最小完成时间剪枝策略来剪枝，只扩展那些可能产生更好解的节点，避免了对不必要的节点进行搜索。  全局最优解的更新：在搜索过程中，需要及时更新全局最优解，以便记录当前最优的调度方案。每当找到一个更优的解时，我就更新全局最优解，以确保最终得到的结果是全局最优的。 | | | |